Pierre Weis Xavier Leroy

1er et 2e CYCLES • ÉCOLES D'INGÉNIEURS

Le langage Caml

2e édition

DUNOD

Pierre Weis

Xavier Leroy

LE LANGAGE CAML

Deuxième édition

Copyright 1992, 1993, 2009 Pierre Weis et Xavier Leroy.

Ce texte est distribué sous les termes de la licence Creative Commons BY-NC-SA. Le texte complet de la licence est disponible à l'adresse suivante :

`http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode`

Voici un résumé des droits et conditions de cette licence.

- Vous êtes libres :
  - de reproduire, distribuer et communiquer cette création au public
  - de modifier cette création
- Selon les conditions suivantes :
  - Paternité. Vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'oeuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'oeuvre).
  - Pas d'Utilisation Commerciale. Vous n'avez pas le droit d'utiliser cette création à des fins commerciales.
  - Partage des Conditions Initiales à l'Identique. Si vous modifiez, transformez ou adaptez cette création, vous n'avez le droit de distribuer la création qui en résulte que sous un contrat identique à celui-ci.
- A chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. La meilleure manière de les indiquer est un lien la page Web ci-dessus.
- Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette oeuvre.
- Rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur ou des auteurs.

À mes parents,
À Suzanne et Michel,
À Lise, Marie, Jean-Baptiste et Irène,
À Hélène.

Pierre Weis

Table des matières

Avant-propos **xi**

I Programmer en Caml **1**

Avertissement **3**

1 Premiers pas **5**
1.1 Idées générales sur Caml 5
1.2 Dialoguer avec Caml 6
1.3 Les définitions 6
1.4 Fonctions 8
1.5 Valeurs et programmes 13
1.6 Impression 13
1.7 Conventions syntaxiques 15
1.8 Diagrammes syntaxiques 17

2 Récursivité **19**
2.1 Fonctions récursives simples 19
2.2 Définitions par cas : le filtrage 27
2.3 Les tours de Hanoi 28
2.4 Notions de complexité 31

3 Programmation impérative **37**
3.1 La programmation impérative 37
3.2 Boucles 39
3.3 Manipulation de polynômes 40
3.4 Impression des polynômes 42
3.5 Caractères et chaînes de caractères 46
3.6 Les références 47
3.7 Un programme utilisant des références 49
3.8 Récursivité et boucles 50
3.9 Règle d'extensionnalité 52
3.10 Effets et évaluation 53

4 Fonctionnelles et polymorphisme **57**
4.1 Notion de polymorphisme 57
4.2 Fonctions d'ordre supérieur 59
4.3 Typage et polymorphisme 61
4.4 Curryfication 64

4.5 Une fonctionnelle de tri polymorphe 65
4.6 La pleine fonctionnalité 67
4.7 Composition de fonctions 70

5 Listes 75
5.1 Présentation 75
5.2 Programmation assistée par filtrage 77
5.3 Tri par insertion 78
5.4 Fonctionnelles simples sur les listes 81
5.5 Les polynômes creux 83
5.6 Filtrage explicite 84
5.7 Opérations sur les polynômes creux 85
5.8 Animation des tours de Hanoi 88
5.9 Fonctionnelles complexes sur les listes 91
5.10 Efficacité des fonctions sur les listes : étude de cas 98
5.11 Listes et récurrence 103
5.12 À la recherche de l'itérateur unique 105

6 Les structures de données 109
6.1 Polynômes pleins et polynômes creux 109
6.2 Types sommes élaborés 113
6.3 Les types somme 116
6.4 Les types produit 116
6.5 Mélange de types somme et types produit 118
6.6 Structures de données mutables 118
6.7 Structures de données et filtrage 120
6.8 Structures de données et récurrence 122

7 Le docteur 125
7.1 Vue d'ensemble 125
7.2 Les exceptions 126
7.3 Fonctions de recherche dans les listes 130
7.4 Traitements de chaînes de caractères 133
7.5 Camélia 135
7.6 Dialogue avec l'utilisateur 140
7.7 Exemple de session 143
7.8 Pour aller plus loin 144

8 Graphisme 147
8.1 Fractales 147
8.2 Le graphisme de Caml 148
8.3 Les nombres en représentation flottante 149
8.4 Le crayon électronique 149
8.5 Premiers dessins 152
8.6 Le flocon de von Koch 154

9 Syntaxe abstraite, syntaxe concrète 155
9.1 Présentation 155
9.2 Le retard à l'évaluation 156

9.3 L'évaluation des ordres du langage graphique 158
9.4 Syntaxe et sémantique 159
9.5 Notions d'analyses syntaxique et lexicale 160
9.6 Analyse lexicale et syntaxique 161
9.7 Ajout des procédures 168

10 Programmes indépendants et modules 179
10.1 Chargement de fichiers 179
10.2 Programmes indépendants 180
10.3 Entrées-sorties de base 181
10.4 Programmes en plusieurs modules 183
10.5 Interfaces de modules 187
10.6 Compilations interactives 190

11 Interfaces graphiques 193
11.1 Structure d'une interface graphique 193
11.2 Relier des composants entre eux 194
11.3 Un convertisseur de devises 196
11.4 Le jeu du taquin 199
11.5 Pour aller plus loin 201

II Exemples complets 203

Avertissement 205

12 Démonstration de propositions 207
12.1 La logique mathématique 207
12.2 Calculs de tables de vérité 210
12.3 Le principe des démonstrations 212
12.4 Représentation et vérification des propositions 213
12.5 Syntaxe concrète des propositions 217
12.6 Le vérificateur de tautologies 221
12.7 Exemples de théorèmes 223
12.8 Pour aller plus loin : l'analyseur lexical universel 228
12.9 Pour aller encore plus loin : le hachage 232

13 Compression de fichiers 237
13.1 La compression de données 237
13.2 Plan du programme 238
13.3 L'algorithme de Huffman 240
13.4 Annexes 247
13.5 Mise en pratique 252
13.6 Pour aller plus loin 252

14 Simulation d'un processeur 255
14.1 Le pico-processeur 255
14.2 Le simulateur 260
14.3 L'assembleur 267
14.4 Pour aller plus loin 275

15 Compilation de mini-Pascal **277**
15.1 Syntaxe abstraite, syntaxe concrète 277
15.2 Typage 283
15.3 Compilation 289
15.4 Pour aller plus loin 304

16 Recherche de motifs dans un texte **305**
16.1 Les motifs 305
16.2 Syntaxe abstraite et syntaxe concrète des motifs 306
16.3 Les automates 309
16.4 Des expressions rationnelles aux automates 310
16.5 Déterminisation de l'automate 313
16.6 Réalisation de la commande `grep` 319
16.7 Annexe 320
16.8 Mise en pratique 321
16.9 Pour aller plus loin 321

III Introspection **323**

17 Exécution d'un langage fonctionnel **325**
17.1 Le langage mini-Caml 325
17.2 L'évaluateur 326
17.3 La boucle d'interaction 331
17.4 Mise en œuvre 333
17.5 Pour aller plus loin 334
17.6 Annexe 336

18 Un synthétiseur de types **339**
18.1 Principes de la synthèse de types 339
18.2 L'algorithme de synthèse de types 344
18.3 Représentation des types 348
18.4 L'unification 353
18.5 Inconnues, généralisation et spécialisation 356
18.6 Impression des types 357
18.7 La boucle d'interaction 358
18.8 Mise en œuvre 359
18.9 Pour aller plus loin 360

19 En guise de conclusion **365**
19.1 Une méthodologie de programmation 365
19.2 La compilation de Caml 367

Index **373**

Avant-propos

On prononce Caml avec le « ca » de café et le « mel » de melba.

AML est un langage de programmation de conception récente qui réussit à être à la fois très puissant et cependant simple à comprendre. Issu d'une longue réflexion sur les langages de programmation, Caml s'organise autour d'un petit nombre de notions de base, chacune facile à comprendre, et dont la combinaison se révèle extrêmement féconde. La simplicité et la rigueur de Caml lui valent une popularité grandissante dans l'enseignement de l'informatique, en particulier comme premier langage dans des cours d'initiation à la programmation. Son expressivité et sa puissance en font un langage de choix dans les laboratoires de recherche, où il a été utilisé pour traiter des problèmes parmi les plus ardus de l'informatique : démonstration assistée par ordinateur, analyses automatique de programmes, systèmes de réécriture, compilation et métacompilation. En bref, Caml est un langage facile avec lequel on résout des problèmes difficiles.

Longtemps réservé à de grosses machines coûteuses, le langage Caml est maintenant disponible gratuitement sur toute une gamme de machines, du micro-ordinateur personnel (PC, Macintosh, ...) aux stations de travail les plus puissantes, ce qui le rend accessible à un vaste public, de l'amateur curieux au professionnel chevronné en passant par l'étudiant informaticien. À ce vaste public, Caml apporte une nouvelle approche de la programmation, des plus fructueuses. L'investissement que vous ferez en apprenant Caml ne sera pas vain : vous constaterez que le langage vous ouvre des horizons nouveaux et qu'il est assez puissant pour que vous y exprimiez simplement des idées complexes. Ce qui se conçoit bien s'énonce clairement et les programmes pour le dire vous viennent aisément en Caml.

Ce livre se propose donc de faire découvrir Caml à *tous ceux qui s'intéressent à la programmation*. Nous nous sommes efforcés d'écrire un livre accessible à tout « honnête homme », mais qui permette cependant de maîtriser le langage et d'en saisir les beautés. Pour ce faire, nous avons combiné une introduction progressive aux principaux traits du langage avec un véritable cours de programmation, illustré de très nombreux exemples de programmes qui vous permettront de saisir comment on utilise Caml et de vous approprier petit à petit ce merveilleux outil. Les exemples vont jusqu'au développement de programmes complets et d'une longueur respectable. Nous nous efforçons de justifier ces exemples, en les replaçant dans leur contexte et en analysant la clarté et l'efficacité

des solutions proposées. Cet ouvrage s'organise comme suit :

- La partie I, « Programmer en Caml », introduit progressivement les traits du langage et les méthodes essentielles de programmation en Caml.
- La partie II, « Exemples complets », montre comment résoudre en Caml un certain nombre de problèmes réalistes de programmation.
- La partie III, « Introspection », ébauche une implémentation de Caml en Caml, expliquant ainsi le typage et l'évaluation de Caml.

En complément de ce livre, les auteurs ont écrit un second ouvrage, intitulé *Manuel de référence du langage Caml* et publié par le même éditeur, contenant tout ce qui est nécessaire au programmeur Caml expérimenté : un manuel de référence du langage Caml et un manuel d'utilisation du système Caml Light, le compilateur Caml que nous utilisons dans ce livre. Les deux livres sont conçus pour être utilisés ensemble : le présent ouvrage renvoie au manuel de référence pour une description exhaustive du langage et des explications détaillées de certains points techniques ; le manuel de référence suppose connues les notions introduites dans cet ouvrage.

Tous les exemples de ce livre sont présentés dans le système Caml Light, un environnement de programmation en Caml fonctionnant à la fois sur micro-ordinateurs (Macintosh et PC) et sur mini-ordinateurs et stations de travail Unix. Il existe d'autres implémentations du langage Caml, comme par exemple Objective Caml, qui ajoute à Caml Light des objets et des classes, ainsi qu'un système de modules plus puissant. L'essentiel de ce qui est dit dans ce livre porte sur le langage et s'applique donc à toutes les implémentations. Nous signalerons les quelques points spécifiques au système Caml Light. Les lecteurs qui souhaitent consulter la documentation complète du système Caml Light peuvent se reporter au *Manuel de référence du langage Caml*, ou à notre site Web `http://caml.inria.fr/`.

Le système Caml Light est distribué gratuitement et peut être reproduit librement à des fins non commerciales. Pour ceux qui ont accès au réseau Internet, Caml Light est disponible sur le Web à l'adresse `http://caml.inria.fr/`. L'Institut National de Recherche en Informatique et en Automatique (INRIA) en assure également la distribution sur cédéroms. Pour obtenir ce cédérom, reportez-vous à l'encadré qui figure en page de copyright.

Nous encourageons le lecteur à se procurer le système Caml Light et à l'installer sur sa machine, suivant les instructions données par exemple dans le chapitre 12 du Manuel de référence. Il pourra ainsi essayer les exemples et expérimenter par lui-même, ce qui lui facilitera grandement la lecture de ce livre.

Remerciements

Nous tenons à remercier Christian Queinnec, Bernard Serpette et Gérard Huet qui se sont astreints à relire ce livre, Valérie Ménissier-Morain qui a participé à l'illustration, Ian Jacobs pour son assistance typographique et Christian Rinderknecht qui a restauré les lettrines, une calligraphie anglaise du huitième siècle. Le jeu de taquin de la section 11.4 est dû à François Rouaix ; l'exemple de la section 11.2 est traduit d'un programme de John Ousterhout.

I

Programmer en Caml

Avertissement

La première partie de ce livre est une introduction progressive au langage Caml. On n'y suppose pas de connaissances préalables autres que des notions élémentaires de mathématiques du niveau du lycée. Les exemples de programmes que nous vous présentons vont de l'exemple d'une ligne au vrai programme de plusieurs pages. Tous les exemples ont été mûrement réfléchis pour être soit étonnants (voire amusants, pourquoi pas ?) soit réellement utiles ou représentatifs des programmes qu'on écrit *vraiment*. Si bien que nous espérons que tous pourront nous lire avec profit, du débutant en programmation, ignorant complètement Caml et désirant s'en faire une idée, à l'étudiant confirmé qui trouvera matière à réflexion dans des programmes non triviaux.

En s'adressant à un si vaste public, nous avons tenté d'accélérer la lecture de tous : le débutant verra souvent des sections qu'on lui suggère de ne pas lire, car elles sont compliquées et pas indispensables pour la suite, tandis que le spécialiste sera invité à sauter des chapitres entiers si ses connaissances le lui permettent. Par exemple, le prochain chapitre débute par un avertissement au spécialiste :

> Si vous savez déjà que « `2 + 2 ; ;` » font « `- : int = 4` », ... , vous pouvez sauter ce chapitre.

En revanche, le chapitre 3 contient une section « Effets et évaluation », qui s'ouvre par un avertissement au débutant : « Cette section peut être sautée en première lecture. »

La démarche que nous avons adoptée, c'est-à-dire l'apprentissage par des exemples intéressants, nous a conduits à présenter les notions du langage *par nécessité* : nous les expliquons lorsqu'elles interviennent et uniquement là. Il se peut donc que certaines notions, inutiles à nos programmes, ne soient pas passées en revue. Cela indique clairement qu'elles ne sont pas essentielles. Si l'on désire absolument une vue exhaustive des possibilités de Caml, on consultera le *Manuel de référence du langage Caml* auquel nous avons déjà fait allusion.

1
Premiers pas

Où l'on vérifie que 2 et 2 font 4.

Si vous savez déjà que « `2 + 2 ; ;` » font « `- : int = 4` » et que « `let f = function x ->` ... » signifie « `let f x =` ... », vous pouvez sauter ce chapitre. Sinon, il vous initiera à l'interaction avec Caml.

1.1 Idées générales sur Caml

Caml est un langage simple : il y a peu de constructions mais ces constructions sont les plus générales possibles. Caml utilise des notations intuitives ou consacrées par l'usage et souvent proches de celles des mathématiques. Par exemple, pour ajouter 1 et 2, il suffit d'écrire `1 + 2`. Et les chaînes de caractères, c'est-à-dire les textes qui ne doivent pas être interprétés par le langage, sont écrites entre des guillemets `"`, notation classique en informatique.

Bien que réalisé en France, Caml est anglophone : ses mots-clés sont en anglais. Ainsi, les valeurs de vérité de la logique mathématique, le vrai et le faux, deviennent `true` et `false` en Caml. Ce n'est pas une réelle difficulté, car les mots-clés sont peu nombreux et nous les traduirons au fur et à mesure.

Caml apporte une grande aide au programmeur, en s'efforçant de détecter le plus possible d'erreurs : le langage analyse les programmes qui lui sont soumis pour vérifier leur cohérence avant toute tentative de compilation ou d'exécution. La principale analyse de cohérence qu'il effectue se nomme le *typage*, mécanisme qui vérifie que les opérations qu'on utilise sont déjà définies et que les valeurs qu'on leur applique ont un sens. Par exemple, l'addition n'est définie que pour les nombres, pas pour les valeurs de vérité ni pour les chaînes de caractères. Donc `true + 1` sera rejeté, de la même façon que `1 + "oui"`. Vous constaterez vite qu'il est ainsi plus difficile d'écrire en Caml des programmes manifestement faux : le langage les rejette automatiquement. Le corollaire est évidemment qu'il est plus facile d'écrire des programmes corrects !

Si vous êtes familier avec un langage algorithmique classique, comme Pascal par exemple, vous ne serez pas complètement dépaysé par Caml : vous y retrouverez la notion de fonction et une notion similaire à celle de procédure ; d'autre part nous avons

déjà vu que Caml est un langage typé. Ces notions sont simplement généralisées et simplifiées : par exemple le typage est automatique et ne nécessite pas d'annotations dans les programmes comme c'est le cas en Pascal.

1.2 Dialoguer avec Caml

Caml offre non seulement un compilateur traditionnel, qui transforme des fichiers de code source en code compilé exécutable par la machine, mais aussi un système interactif qui vous permet de dialoguer directement avec Caml, sans passer par l'intermédiaire d'un fichier. Dans ce mode, le langage s'utilise comme une « calculette » : vous tapez des phrases au clavier de l'ordinateur et Caml réagit en vous donnant immédiatement les résultats de vos programmes. Nous utiliserons d'abord cette méthode d'interaction directe car elle facilite l'apprentissage. Nous verrons plus tard l'utilisation du compilateur indépendant, à partir du chapitre 10. Vous pouvez donc entrer au terminal les exemples qui suivent, si vous avez déjà installé le système Caml Light sur votre machine.

Toutes les phrases soumises à Caml doivent être munies d'une indication de fin de phrase, ce qu'on note en Caml par `;;` (deux points-virgules accolés). C'est justifié pour un système qui offre une version interactive, dans la mesure où il est impossible de deviner quand l'utilisateur a terminé sa phrase : par exemple après `1 + 2`, il est permis d'écrire encore `+ 3` pour évaluer `1 + 2 + 3`. D'autre part, une phrase peut s'étendre sur autant de lignes que nécessaire ; la fin de la ligne ne se confond donc pas avec la fin de la phrase. On devra donc écrire `;;` pour signaler la fin de la phrase. Il faut bien entendu également appuyer sur la touche « retour chariot » (*return* en anglais) comme c'est traditionnel pour toute interaction avec un ordinateur.

En réponse au signe d'invite de Caml (le caractère `#` que Caml imprime pour indiquer qu'il attend que nous tapions quelque chose), demandons-lui d'effectuer un petit calcul : l'addition de 2 et de 2. Pour cela nous entrons simplement l'opération à effectuer, `2 + 2`, suivie de la marque de fin de phrase `;;`.

```
# 2 + 2;;
- : int = 4
```

Caml nous répond immédiatement, en indiquant par un signe `-` que nous avons simplement calculé une *valeur*, que cette valeur est de type entier (`: int`) et qu'elle vaut 4 (`= 4`). Vous constatez que Caml a déduit tout seul le type du résultat du calcul. Pour un exemple si simple, ce n'est pas vraiment impressionnant, mais c'est un mécanisme absolument général : quelle que soit la complexité du programme que vous lui soumettrez, Caml en déduira le type sans aucune intervention de votre part.

1.3 Les définitions

Vous pouvez donner un nom à une valeur que vous calculez, pour ne pas perdre le résultat de votre calcul. La construction qui permet ainsi de nommer des valeurs s'appelle une *définition*.

Définitions globales

De même qu'en mathématiques on écrit : « soit s la somme des nombres 1, 2 et 3 », on écrit en Caml (« soit » se traduit par *let* en anglais) :

```
# let s = 1 + 2 + 3;;
s : int = 6
```

Caml nous répond que nous avons défini un nouveau nom `s`, qui est de type entier (`: int`) et vaut 6 (`= 6`). Maintenant que le nom `s` est défini, il est utilisable dans d'autres calculs ; par exemple, pour définir le carré de `s`, on écrirait :

```
# let s2 = s * s;;
s2 : int = 36
```

Les définitions sont des *liaisons* de noms à des valeurs. On peut considérer ces noms (qu'on appelle aussi identificateurs, ou encore variables) comme de simples abréviations pour la valeur qui leur est liée. En particulier, une définition n'est pas modifiable : un nom donné fait toujours référence à la même valeur, celle qu'on a calculée lors de la définition du nom. Le mécanisme du « `let` » est donc fondamentalement différent du mécanisme d'affectation, que nous étudierons plus loin. Il est impossible de changer la valeur liée à un nom ; on peut seulement *redéfinir* ce nom par une nouvelle définition, donc un nouveau « `let` ».

Une fois défini, un nom a toujours la même valeur

Informatique et mathématiques

La grande différence entre les mathématiques et les langages de programmation, même ceux qui se rapprochent des mathématiques comme Caml, est qu'un langage de programmation calcule avec des valeurs et non pas avec des quantités formelles. Par exemple, en mathématiques, si x est un entier, alors $x - x$ est égal à 0. Il n'est pas nécessaire de connaître la *valeur* de x pour obtenir le résultat du calcul ; on parle d'ailleurs plutôt de *simplification* que de calcul. Au contraire, en Caml, on ne peut parler d'un nom s'il n'a pas été précédemment défini. On ne peut donc pas calculer $x - x$ si x n'est pas lié à une valeur précise, car il est alors impossible de faire effectivement la soustraction :

```
# x - x;;
Entrée interactive:
>x - x;;
>^
L'identificateur x n'est pas défini.
```

Le langage indique ici que notre phrase est erronée puisque le nom `x` n'a jamais été défini ; on dit encore que `x` n'est pas « lié ». Mais le résultat est évidemment le bon, dès que l'identificateur est défini ; nous pouvons par exemple faire le calcul avec `s` :

```
# s - s;;
- : int = 0
```

Une autre différence essentielle entre un programme et une définition mathématique réside dans la notion d'efficacité : un langage de programmation calcule des valeurs de

manière effective, ce qui demande un certain temps. Si ce temps devient prohibitif, on peut considérer que le programme est incorrect, même si l'on peut prouver qu'il donnerait un jour le bon résultat. En mathématiques, cette notion de temps de calcul est sans importance. Un autre écueil majeur de l'informatique est qu'elle ne connaît pas l'infini. Par exemple, la limite quand n tend vers l'infini de $f(n)$, ce qu'on note en mathématiques $\lim_{n\to\infty} f(n)$ et qui signifie la valeur de $f(n)$ quand n devient arbitrairement grand, existe éventuellement en mathématiques, mais ne peut qu'être approchée par une machine. Enfin, l'évaluation des expressions d'un langage de programmation tel que Caml ne *termine* pas toujours : les calculs peuvent « boucler » et donc ne jamais s'achever. Autrement dit, les fonctions définissables par un programme sont en général des fonctions *partielles* (non définies pour certaines valeurs) plutôt que des fonctions *totales* (toujours définies).

Définitions locales

Les définitions de noms que nous venons de voir sont permanentes : elles restent valides tant que vous n'abandonnez pas le système Caml. Ces définitions « définitives » sont qualifiées de *globales*. Cependant, pour faire un petit calcul, il est inutile d'utiliser des définitions globales : on dispose donc en Caml d'un moyen de définir temporairement des noms, pour la seule durée du calcul en question. Ces définitions temporaires sont les définitions *locales*, qui disparaissent à la fin de l'évaluation de la phrase dans laquelle elles se trouvent. Ces définitions locales ne sont donc plus valides après le calcul de l'expression qui les suit (après le mot-clé `in`, qui signifie « dans ») :

```
# let s = 20 in s * 4;;
- : int = 80
```

Le nom `s` a été lié à `20` pendant le calcul de `s * 4`, mais la définition précédente de `s` reste inchangée. Pour s'en rendre compte, il suffit de demander la valeur de `s`, c'est-à-dire le résultat du calcul réduit à `s` :

```
# s;;
- : int = 6
```

La définition locale d'un nom est complètement indépendante du type actuel du nom : par exemple, `s` et `s2` qui sont actuellement de type `int` peuvent être définis localement avec le type `string` :

```
# let s = "Le langage " and s2 = "Caml" in s ^ s2;;
- : string = "Le langage Caml"
```

Cet exemple utilise l'opérateur `^` qui met deux chaînes de caractères bout à bout (concaténation). Notez également que les définitions multiples consistent en une simple succession de définitions séparées par le mot-clé `and` (qui signifie « et »).

1.4 Fonctions

Les fonctions forment les constituants élémentaires des programmes en Caml. Un programme n'est rien d'autre qu'une collection de définitions de fonctions, suivie d'un appel à la fonction qui déclenche le calcul voulu.

Définir une fonction

Définir une fonction en Caml est simple et naturel, car la syntaxe est très proche des notations mathématiques usuelles. À la définition mathématique « soit *successeur* la fonction définie par $successeur(x) = x + 1$ » correspond la définition Caml suivante :

```
# let successeur (x) = x + 1;;
successeur : int -> int = <fun>
```

Caml nous indique encore une fois que nous avons défini un nom : `successeur`. Ce nom a pour type `int -> int` (`->` se prononce « flèche »), qui est le type des fonctions des entiers (`int`) vers les entiers (`-> int`) et ce nom a pour valeur une fonction (`= <fun>`). Le système a trouvé tout seul le type de la fonction, mais il ne sait pas comment imprimer les valeurs fonctionnelles, parce que leur représentation interne est faite de code machine ; il affiche donc simplement `<fun>` sans plus de précisions. Effectivement, le nom `successeur` possède maintenant une valeur :

```
# successeur;;
- : int -> int = <fun>
```

Une définition de fonction n'est donc pas essentiellement différente d'une définition d'entier ou de chaîne de caractères. Elle définit simplement le nom de la fonction et lui donne une valeur qui est une fonction, ce qu'on appelle une *valeur fonctionnelle*.

Application de fonctions

L'application d'une fonction à son argument suit aussi la convention mathématique (rappelons que « $f(x)$ » se prononce f *de* x) :

```
# successeur (2);;
- : int = 3
```

Le langage Caml fournit une syntaxe plus souple pour utiliser et définir les fonctions : on peut supprimer les parenthèses autour des noms des arguments des fonctions aussi bien au cours d'une définition que lors d'une application. Étant donnée la paresse légendaire des programmeurs, c'est bien sûr cette habitude qui prédomine ! Avec cette convention, on écrit simplement

```
# let successeur x = x + 1;;
successeur : int -> int = <fun>
# successeur 2;;
- : int = 3
```

Définitions locale de fonctions

Rien n'empêche de définir une fonction localement, bien que cela surprenne souvent les débutants en Caml. Voici un exemple de fonction locale :

```
# let prédécesseur x = x - 1 in
  (prédécesseur 3) * (prédécesseur 4);;
- : int = 6
```

La fonction `prédécesseur` n'est définie que pendant le calcul du produit des prédécesseurs de 3 et 4.

Les définitions locales sont aussi utilisées dans les définitions globales, par exemple pour calculer la formule qui définit une fonction (ce qu'on appelle le *corps* de la fonction). Définissons par exemple la fonction `prédécesseur_carré` qui retourne le carré du prédécesseur d'un nombre (la fonction $x \mapsto (x-1)^2$). Nous définissons localement le prédécesseur de l'argument, puis l'élevons au carré :

```
# let prédécesseur_carré x =
    let prédécesseur_de_x = x - 1 in
    prédécesseur_de_x * prédécesseur_de_x;;
prédécesseur_carré : int -> int = <fun>
# prédécesseur_carré 3;;
- : int = 4
```

Une fonction peut aussi définir localement une autre fonction. Par exemple, pour définir la fonction `puissance4` qui élève son argument à la puissance quatre, il est naturel d'utiliser la formule $x^4 = (x^2)^2$, donc d'élever au carré le carré de l'argument. Pour cela, on définit localement la fonction `carré` et on l'utilise deux fois :

```
# let puissance4 x =
    let carré y = y * y in (* définition locale d'une fonction *)
    carré (carré x);;
puissance4 : int -> int = <fun>
# puissance4 3;;
- : int = 81
```

Comme on le voit sur cet exemple, les commentaires en Caml sont encadrés entre `(*` et `*)`. Ils peuvent contenir n'importe quel texte, y compris d'autres commentaires, et s'étendre sur plusieurs lignes.

Fonctions à plusieurs arguments

Les fonctions possédant plusieurs arguments ont simplement plusieurs noms d'arguments dans leur définition :

```
# let moyenne a b = (a + b) / 2;;
moyenne : int -> int -> int = <fun>
# let périmètre_du_rectangle longueur largeur =
      2 * (longueur + largeur);;
périmètre_du_rectangle : int -> int -> int = <fun>
```

Le type de ces deux fonctions, `int -> int -> int`, indique qu'elles prennent deux arguments de type `int` (`int -> int ->`) et calculent un entier (`-> int`).

Lorsque des fonctions ont plusieurs arguments, il faut évidemment leur fournir aussi leur compte d'arguments quand on les applique. Ainsi, un appel à `périmètre_du_rectangle` ou `moyenne` comportera deux arguments :

```
# périmètre_du_rectangle 3 2;;
- : int = 10
# moyenne 5 3;;
- : int = 4
```

Fonctions anonymes

Une fonction Caml est un « citoyen à part entière », on dit aussi « citoyen de première classe », c'est-à-dire une valeur comme toutes les autres. Une fonction a le même statut qu'un nombre entier : elle est calculée, on peut la passer en argument ou la retourner en résultat. Les valeurs fonctionnelles sont créées lors des définitions de fonctions, comme nous venons de le voir. Cependant, on peut aussi construire des valeurs fonctionnelles sans leur donner de nom, en utilisant des *fonctions anonymes*. Ces fonctions sont introduites par le mot-clé `function`, suivi de la formule qui les définit :

```
# (function x -> 2 * x + 1);;
- : int -> int = <fun>
```

Encore une fois, Caml nous indique par le symbole `-` que nous avons fait un simple calcul, dont le résultat est de type `int -> int` et dont la valeur est une fonction (`= <fun>`). On applique les fonctions anonymes comme toutes les autres fonctions, en les faisant suivre de leur(s) argument(s) :

```
# (function x -> 2 * x + 1) (2);;
- : int = 5
```

Définition de fonctions à l'aide de fonctions anonymes

Il existe un autre style de définitions mathématiques de fonctions :

$$\begin{array}{rccc} \text{« Soit } \textit{successeur} \ : & \mathbb{Z} & \to & \mathbb{Z} \\ & x & \mapsto & x+1 \text{ »} \end{array}$$

Ce style insiste sur le fait que *successeur* est une fonction qui à tout élément x de l'ensemble $\mathbb{Z}$ des entiers associe $x+1$. À l'aide des fonctions anonymes, cette définition se traduit très simplement en Caml :

```
# let successeur = function x -> x + 1;;
successeur : int -> int = <fun>
```

Contraintes de type

Pour se rapprocher encore du style de la définition mathématique, on peut même ajouter une contrainte de type sur le nom de la fonction, qui rend compte de l'indication $\textit{successeur} : \mathbb{Z} \to \mathbb{Z}$ des mathématiques. Une *contrainte de type* (ou annotation de type) est une indication explicite du type d'une expression Caml. Vous pouvez, si vous le souhaitez, ajouter des annotations de type dans vos programmes, par exemple pour aider à la relecture. Pour annoter un morceau de programme avec un type, il suffit de mettre ce morceau de programme entre parenthèses avec son type, avec la même convention que le système interactif, c'est-à-dire un « `:` » suivi d'un nom de type :

```
# ("Caml" : string);;
- : string = "Caml"
```

Nous obtenons maintenant une définition de la fonction `successeur` très fidèle à celle des mathématiques :

```
# let (successeur : int -> int) = function x -> x + 1;;
successeur : int -> int = <fun>
```

Ce style revient à définir le nom `successeur` comme un nom ordinaire, mais dont la valeur est une fonction. Cette définition est absolument équivalente à la précédente définition de `successeur` :

```
let successeur (x) = x + 1;;
```

Fonctions anonymes à plusieurs arguments

Le choix entre les deux modes de définition des fonctions est donc, comme en mathématiques, une simple affaire de style. En règle générale, le style « `let successeur (x) =` » est plus concis, particulièrement lorsque la fonction a plusieurs arguments, puisque l'autre style oblige à introduire chacun des arguments par une construction « `function argument ->` ». Par exemple, définir la fonction `moyenne` dans le style « `function x ->` » conduirait à écrire :

```
# let moyenne = function a -> function b -> (a + b) / 2;;
moyenne : int -> int -> int = <fun>
```

Au passage, nous remarquons qu'une fonction anonyme a parfaitement le droit d'avoir plusieurs arguments. Attention : il n'est pas permis d'écrire `function a b ->`, il faut impérativement répéter le mot-clé `function`, une fois par argument. C'est pourquoi nous utiliserons la plupart du temps le style le plus léger, celui qui évite d'employer le mot `function`.

Les tests et l'alternative

Caml fournit une construction pour faire des calculs qui dépendent d'une condition : c'est l'*alternative*, le classique « `if` ... `then` ... `else` ... ». Cette construction correspond au calcul « si *condition* alors $expression_1$ sinon $expression_2$ », qui signifie simplement qu'il faut calculer $expression_1$ si la condition est vraie et $expression_2$ sinon. Nous illustrons cette construction en *implémentant* (c'est-à-dire en réalisant sur machine) la fonction « valeur absolue », qui calcule la valeur d'un nombre indépendamment de son signe. Cette fonction, notée $|x|$ en mathématiques, est définie comme :

$$|x| = \begin{cases} x & \text{si } x \geq 0 \\ -x & \text{sinon} \end{cases}$$

Sachant qu'en Caml les comparaisons entre nombres entiers suivent les notations mathématiques (`<`, `>`, `=`, `>=` et `<=`), nous sommes armés pour définir la fonction valeur absolue :

```
# let valeur_absolue (x) = if x >= 0 then x else -x;;
valeur_absolue : int -> int = <fun>
# valeur_absolue (3);;
- : int = 3
# valeur_absolue (-3);;
- : int = 3
```

Valeurs de vérité

Remarquons que les tests calculent un résultat, une *valeur de vérité*. Une valeur de vérité est soit « vrai », soit « faux », ce qui se note `true` et `false` en Caml. On appelle aussi les valeurs de vérité « valeurs booléennes », en l'honneur du logicien Boole ; elles sont du type `bool`. On peut donc employer les tests pour calculer un booléen :

```
# 2 < 1;;
- : bool = false
# (valeur_absolue (3)) = (valeur_absolue (-3));;
- : bool = true
```

1.5 Valeurs et programmes

Nous venons de faire des calculs. Mais où sont donc les programmes ? Ce sont tout simplement les fonctions ! Un programme consiste en une définition de fonction qui calcule le résultat désiré. En général cette fonction utilise à son tour d'autres fonctions, qui correspondent à la notion de sous-programmes. Par exemple, si vous désirez calculer la somme des carrés de deux nombres, vous définirez d'abord la fonction `carré` :

```
# let carré (x) = x * x;;
carré : int -> int = <fun>
```

pour ensuite définir la fonction désirée :

```
# let somme_des_carrés x y = carré (x) + carré (y);;
somme_des_carrés : int -> int -> int = <fun>
```

et enfin l'appliquer dans le cas qui vous intéresse :

```
# somme_des_carrés 3 4;;
- : int = 25
```

En résumé : une fonction manipule des valeurs (entiers, chaînes de caractères, booléens) qui ont toutes un type ; la fonction elle-même est une valeur et possède donc un type. En ce sens, les programmes en Caml sont des valeurs !

1.6 Impression

Notion d'effet

Caml propose bien sûr le moyen d'imprimer des valeurs à l'écran ou dans des fichiers. On utilise pour cela des fonctions dont le but n'est pas d'effectuer des calculs mais de produire des *effets*, c'est-à-dire une action sur le monde extérieur, par exemple une interaction avec l'utilisateur du programme, l'écriture d'un fichier ou d'un message au terminal.

Un premier effet

Nous allons réaliser un premier effet très simple : nous écrivons « Bonjour ! » à l'écran en utilisant la fonction prédéfinie `print_string` qui a justement pour effet d'imprimer son argument au terminal. Une *fonction prédéfinie* est une fonction qui

vous est fournie par le système Caml ; vous n'avez donc pas besoin de l'écrire. Ces fonctions sont décrites en détails dans le *Manuel de référence du langage Caml*. Elles sont aussi appelées « fonctions primitives » ou tout simplement « primitives ». Essayons la primitive `print_string` :

```
# print_string "Bonjour!";;
Bonjour!- : unit = ()
```

L'impression s'est produite comme prévu. Cependant Caml nous indique aussi que nous avons calculé un résultat de type `unit` et qui vaut `()`. Le type `unit` est un type prédéfini qui ne contient qu'un seul élément, « `()` », qui signifie par convention « rien ». Nous n'avons pas demandé ce résultat : tout ce que nous voulions, c'est faire une impression (un effet). Mais toutes les fonctions Caml doivent avoir un argument et rendre un résultat. Lorsqu'une fonction opère uniquement par effets, on dit que cette fonction est une *procédure* On utilise alors « rien », c'est-à-dire `()`, en guise de résultat ou d'argument. (En position d'argument dans une définition de fonction, on peut considérer `()` comme un argument minimal : l'argument `(x)` auquel on aurait même retiré la variable `x` ; de même en résultat, `()` figure une expression parenthésée dont tout le texte aurait disparu.)

Impressions successives : séquencement

Supposez qu'il nous faille imprimer deux textes successifs à l'écran : par exemple, « Bonjour » puis « tout le monde ! ». Nous devons faire deux effets à la suite l'un de l'autre, *en séquence*. Évaluer en séquence deux expressions e_1 et e_2 signifie simplement les évaluer successivement : d'abord e_1, puis e_2. Comme dans la plupart des langages de programmation, la séquence est notée par un point virgule en Caml. L'opération « e_1 puis e_2 » s'écrit donc e_1 `;` e_2. Nous écrivons donc :

```
# print_string "Bonjour "; print_string "tout le monde!";;
Bonjour tout le monde!- : unit = ()
```

La machine a d'abord imprimé `Bonjour` puis `tout le monde!`, comme on s'y attendait. Le résultat de toute l'opération (de toute la séquence) est « rien ». Cela s'explique naturellement parce que le résultat de la première impression (un premier « rien ») a été oublié. De manière générale, la séquence « jette » le résultat du premier calcul et renvoie le résultat du second : e_1 `;` e_2 s'évalue en la même valeur que e_2. Comme le résultat de e_1 est détruit, il est clair que l'expression e_1 n'est utile que si elle produit des effets : il serait stupide d'évaluer la séquence `(1 + 2); 0` qui rendrait exactement le même résultat que `0`.

```
# (1 + 2); 0;;
Entrée interactive:
>(1 + 2); 0;;
> ^^^^^
Attention: cette expression est de type int,
mais est utilisée avec le type unit.
- : int = 0
```

On constate d'ailleurs que le compilateur émet une alerte pour indiquer que l'expression `(1 + 2)` produit un résultat qui sera ignoré !

Pour délimiter précisément une séquence, on l'encadre souvent entre les mots-clés `begin` (début) et `end` (fin) :

```
# begin
    print_string "Voilà ";
    print_string "Caml!";
    print_newline ()
  end;;
Voilà Caml!
- : unit = ()
```

La dernière expression, `print_newline ()`, fait imprimer un retour chariot. La fonction `print_newline` opère entièrement par effets, elle n'a donc pas de paramètre significatif, ni de résultat significatif.

1.7 Conventions syntaxiques

Résumons quelques conventions syntaxiques qu'il est bon d'avoir en tête pour la suite.

Définitions de fonctions

Pour les définitions de fonctions, nous avons la convention suivante :

`let` f x `=` ... est équivalent à `let` f `=` `function` x `->` ...

On peut itérer cette convention pour définir les fonctions à plusieurs arguments :

`let` f x y `=` ... est équivalent à `let` f `=` `function` x `->` `function` y `->` ...

Application de fonctions

Pour ce qui est de l'application de fonction, nous avons vu que les parenthèses autour de l'argument était facultatives :

Si x est une variable ou une constante, $f\ x$ est équivalent à $f\ (x)$

Attention : cette convention n'est valable que lorsque x est une variable ou une constante. Si vous employez cette convention avec une expression plus compliquée, les parenthèses retrouvent leur sens habituel en mathématiques (le groupement des expressions) et la présence de parenthèses modifie alors le résultat du calcul. Par exemple, l'argument négatif d'une fonction doit impérativement être parenthésé : $f\ (-1)$ sans parenthèses est compris comme la soustraction $f - 1$. De même, si l'argument est une expression complexe, vous ne pouvez pas supprimer les parenthèses sans changer le résultat :

```
# successeur (2 * 3);;
- : int = 7
```

```
# successeur 2 * 3;;
- : int = 9
```

Cette dernière phrase est comprise par Caml comme si vous aviez écrit `(successeur 2) * 3`. C'est d'ailleurs un phénomène général pour tous les opérateurs de Caml : les applications de fonctions en argument des opérations sont implicitement parenthésées. Par exemple `successeur 2 - successeur 3` est lu comme `(successeur 2) - (successeur 3)`, et de même pour tous les opérateurs : `successeur 2 >= successeur 3` est correctement interprété.

$f\ x + g\ y$ est équivalent à $(f\ x) + (g\ y)$

Application de fonctions à plusieurs arguments

L'application de fonctions à plusieurs arguments suit les mêmes conventions : `moyenne (2) (6)` est équivalent à `moyenne 2 6`, mais vous devez conserver les parenthèses si vous voulez calculer `moyenne (2 * 3) (3 + 3)`.

Techniquement, on dit que l'application « associe à gauche » en Caml, ce qui signifie que les parenthèses peuvent être omises dans $(f\ x)\ y$, qui correspond au résultat de f de x appliqué à y, mais qu'elles sont indispensables dans $f\ (g\ x)$, qui signifie au contraire f appliquée au résultat de l'application de la fonction g à x.

$f\ x\ y$ est équivalent à $(f\ x)\ y$

Au lieu de `moyenne 2 6`, on peut donc écrire `(moyenne 2) 6` . La deuxième forme est évidemment un peu étrange, mais elle a la même signification que la première. On a donc beaucoup de manières équivalentes d'exprimer l'application de la fonction `moyenne` à `2` et à `6`. La plus simple est évidemment sans parenthèses aucunes : « `moyenne 2 6` ». Mais l'on peut écrire aussi « `moyenne (2) (6)` » ou, en utilisant la règle précédente pour ajouter encore des parenthèses, « `(moyenne (2)) 6` » ou même « `(moyenne (2)) (6)` ». En pratique, nous utiliserons toujours la forme la plus simple, sans parenthèses.

En revanche, on ne peut absolument pas grouper les arguments `2` et `6` à l'intérieur de parenthèses : `moyenne (2 6)` est erroné. Cela signifierait en effet qu'on désire appliquer `moyenne` à un seul argument « `2 6` ». Qui plus est, cela voudrait dire qu'on tente d'appliquer le nombre 2 au nombre 6 ! Des expressions construites sur le modèle `moyenne (2 6)`, c'est-à-dire, plus généralement, du genre $f\ (g\ y)$, ont pourtant un sens. Considérez, par exemple le calcul du successeur du successeur de 1. On écrit naturellement :

```
# successeur (successeur 1);;
- : int = 3
```

Mais si l'on ôte les parenthèses, on écrit `successeur successeur 1` et cela signifie maintenant que nous voulons appliquer la fonction `successeur` à deux arguments (le premier argument serait la fonction `successeur` elle-même et le second argument serait `1`). Cependant la fonction `successeur` n'admet qu'un seul argument ; si nous retirons les parenthèses (sciemment ou par inadvertance), Caml nous indique donc une erreur :

```
# successeur successeur 1;;
Entrée interactive:
>successeur successeur 1;;
>           ^^^^^^^^^^
Cette expression est de type int -> int,
mais est utilisée avec le type int.
```

Le message indique en effet que l'expression soulignée (`successeur`) est une fonction de type `int -> int` : elle ne peut pas être utilisée comme un argument entier.

Retenons de toute façon que :

$f\ (g\ y)$ n'est pas équivalent à $f\ g\ y$

1.8 Diagrammes syntaxiques

Nous résumons la manière d'écrire les constructions de Caml au moyen de définitions simplifiées de syntaxe, telles que :

$$\begin{array}{rcl} \textit{expression} & ::= & \textit{entier} \\ & | & \textit{chaîne-de-caractères} \\ & | & \textit{booléen} \\ & | & \ldots \end{array}$$

Cette définition signifie qu'une expression du langage Caml (*expression*) est (::=) ou bien un entier (*entier*), ou bien (|) une chaîne de caractères (*chaîne-de-caractères*), ou bien (|) un booléen (*booléen*), ou bien ... Ceci n'est qu'un exemple et nous ne faisons évidemment pas figurer toutes les constructions concernant les expressions. D'ailleurs, nous savons déjà qu'une séquence est une expression ; elle devrait donc figurer dans l'ensemble des expressions, de même que les expressions parenthésées (toute expression entourée de parenthèses est une expression). Avec ce formalisme, une séquence se décrit par :

$$\textit{séquence} ::= \textit{expression}\ \texttt{;}\ \textit{expression}$$

ou encore, si elle est délimitée par les mots-clés `begin` et `end` :

$$\textit{séquence} ::= \texttt{begin}\ \textit{expression}\ \texttt{;}\ \textit{expression}\ \texttt{end}$$

Cette méthode de description de la syntaxe d'un langage est appelée « syntaxe BNF », pour Backus-Naur Form, des noms de John Backus et Peter Naur qui l'ont répandue.

2
Récursivité

Où l'on apprend à parler de ce qu'on ne connaît pas encore ...

Vous ne connaissez pas les fonctions récursives, ou n'y avez jamais rien compris, ou bien vous vous passionnez pour les tours de Hanoi ? Alors lisez ce chapitre, il est fait pour vous. Nous y apprendrons une nouvelle manière d'utiliser l'ordinateur. Jusqu'à présent nous nous en servions pour obtenir les solutions à des problèmes que nous savions résoudre, mais qui demandaient trop de calculs mathématiques pour que nous les traitions à la main. C'est l'emploi de l'ordinateur le plus répandu actuellement : la machine sert à effectuer un très grand nombre de fois des opérations simples (pensez à des logiciels de gestion, paye ou comptes en banque). Avec la récursivité, on élève la programmation à un rang autrement plus noble : on écrit des programmes qui résolvent des problèmes que l'on ne sait pas forcément résoudre soi-même. Parvenir à diriger une machine pour qu'elle trouve pour nous la solution d'un problème est réellement fascinant, il est même grisant de voir apparaître cette solution au terminal, quand elle est le résultat de l'exécution d'un programme qu'on a soi-même écrit sans avoir conscience de savoir résoudre le problème.

2.1 Fonctions récursives simples

Notion de récursivité

Une définition récursive est une définition dans laquelle intervient le nom qu'on est en train de définir. Cela correspond dans le langage courant à une phrase qui « se mord la queue ». L'exemple typique est la réponse à la question « Qu'est-ce qu'un égoïste ? » : « Quelqu'un qui ne pense pas à moi ! ». Il est clair qu'on soupçonne légitimement ce genre de phrases d'être dénuées de sens et que c'est souvent le cas. Pourtant, les définitions récursives sont très employées en mathématiques et en informatique, domaines où l'on se méfie beaucoup des phrases « dénuées de sens ». Il existe donc forcément des phrases qui « se mordent la queue » et pourtant possèdent une signification précise, utilisable en mathématiques ou informatique. Toute la difficulté des définitions récursives provient de la détection des cas où la récursivité entraîne le non-sens. Une définition récursive sensée est qualifiée de *bien fondée*. Nous verrons par l'exemple ce que cela veut dire.

L'intuition la plus simple qu'on puisse donner de la récursivité est l'idée de « recommencer » la même chose. La récursivité est présente aussi dans le domaine graphique, un exemple nous est donné par le célèbre dessin qui orne les couvercles de « Vache qui rit », figurant une vache qui porte en boucles d'oreilles des boîtes de « Vache qui rit » (dont les couvercles comportent donc le dessin lui-même). Dans le domaine physique, l'infinité d'images qui apparaît dans deux miroirs quasi parallèles est aussi une bonne analogie (expérience habituelle chez le coiffeur).

Prenons un exemple plus informatique : la très célèbre fonction « factorielle », qui retourne le produit des nombres entiers inférieurs ou égaux à son argument. En mathématiques, elle est notée par un point d'exclamation (!) placé après son argument. On a par exemple $4! = 4 \times 3 \times 2 \times 1$. La fonction factorielle est définie par :

$$n! \;=\; \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{sinon.} \end{cases}$$

Cette définition est récursive : le nom « ! » intervient dans le corps de sa propre définition. Donc, pour comprendre la signification de $n!$ il faut comprendre celle de $(n-1)!$. Cela semble difficile, car on exige la compréhension d'une notion qu'on est justement en train d'expliquer ... En fait, cela se justifie parce que le calcul de $n!$ termine toujours : il suffit d'être patient et de continuer le calcul jusqu'à atteindre 0, ce qui arrivera forcément puisqu'on explique la signification de $n!$ en fonction de $(n-1)!$. Par exemple :

$$\begin{aligned} 3! &= 3 \times (3-1)! = 3 \times 2! \\ &= 3 \times 2 \times (2-1)! = 3 \times 2 \times 1! \\ &= 3 \times 2 \times 1 \times (1-1)! = 3 \times 2 \times 1 \times 0! \\ &= 3 \times 2 \times 1 \times 1 = 6 \end{aligned}$$

La définition mathématique récursive de la fonction factorielle est donc bien fondée : on obtient finalement un résultat pour tout argument entier naturel.

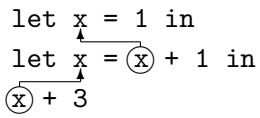
Ce genre de définition se traduit très facilement en Caml ; mais pour cela nous devons explicitement prévenir Caml que nous désirons faire une définition récursive, grâce à une nouvelle construction : `let rec`. Avant de l'employer, expliquons pourquoi il est nécessaire d'introduire une nouvelle construction.

Portée statique et définitions récursives

En mathématiques, vous ne pouvez parler d'une entité mathématique avant d'en avoir donné la définition. Le même axiome est vrai en Caml : vous ne pouvez utiliser un identificateur s'il n'a reçu une définition préalable. C'est ce qu'on appelle la *portée statique*, parce que vous pouvez trouver la définition de n'importe quel identificateur indépendamment du comportement du programme à l'exécution (au contraire de la *portée dynamique*, où la valeur d'un identificateur dépend de la façon dont le calcul se déroule). En Caml, pour une occurrence quelconque d'un identificateur, disons `x`, il vous suffit d'examiner le texte du programme *qui précède* `x` pour trouver la liaison qui a défini `x`. Cette liaison est soit une définition de `x` : `let x =`, soit une définition de fonction qui a `x` pour paramètre : `function x ->`.

```
# let x = 1 in
  let x = x + 1 in
  x + 3;;
- : int = 5
```

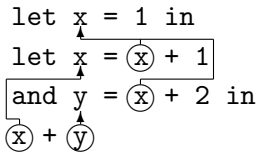
L'identificateur `x` qui apparaît dans `x + 1` dans la définition `let x = x + 1` est lié à `1` (par la précédente définition `let x = 1`), alors que le dernier `x` dans `x + 3` est lié à `2` par la précédente définition `let x = x + 1`, comme le suggère ce schéma :



Le mécanisme de liaison est similaire pour des définitions simultanées (définitions séparées par le mot-clé `and`).

```
# let x = 1 in
  let x = x + 1
  and y = x + 2 in x + y;;
- : int = 5
```

Les deux `x` intervenant dans les définitions `let x = ... and y = ...` font tous les deux référence au nom `x` précédemment défini par `let x = 1`. Les liaisons sont mises en évidence dans ce schéma :

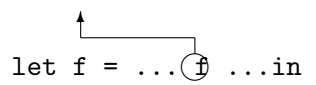


On retiendra que, dans une définition Caml (y compris une définition simultanée),

> Un nom fait toujours référence à une définition préalable.

La construction `let rec`

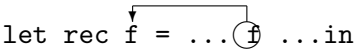
Ceci pose évidemment problème pour définir des fonctions récursives : nous ne pouvons utiliser une définition introduite par un `let`, à cause de la règle de portée statique. En effet, si nous écrivons `let f = ... f ...`, l'occurrence de `f` dans l'expression définissante `... f ...` ne correspond pas au nom `f` que nous sommes en train de définir (en particulier parce que `f` n'est pas encore définie !) mais doit correspondre à une définition *précédente* de `f`. Un petit schéma vaut mieux qu'un long discours : la liaison de `f` s'établit vers le passé.



C'est pourquoi une simple construction `let` ne permet pas de définir une fonction récursive :

```
# let factorielle n = if n = 0 then 1 else n * factorielle (n - 1);;
Entrée interactive:
>let factorielle n = if n = 0 then 1 else n * factorielle (n - 1);;
>                                             ^^^^^^^^^^
L'identificateur factorielle n'est pas défini.
```

En bref : une liaison `let` n'est pas récursive ; il y a donc en Caml une construction spéciale, `let rec`, destinée à introduire les définitions récursives. Cette construction établit une liaison de la forme suivante :



Maintenant, nous définissons facilement la fonction factorielle :

```
# let rec factorielle n =
    if n = 0 then 1 else n * factorielle (n - 1);;
factorielle : int -> int = <fun>
# factorielle 3;;
- : int = 6
```

Compter à l'endroit et à l'envers

Pour comprendre comment s'exécute un appel à une fonction récursive, définissons une fonction qui énumère les nombres par ordre décroissant jusqu'à 1, à partir d'une certaine limite : par exemple, pour une limite de 5 nous voulons obtenir l'impression de « 5 4 3 2 1 » sur l'écran.

Bien que le but de ce programme soit exclusivement de produire des effets, la récursivité s'y introduit naturellement, puisque énumérer à partir d'une certaine limite n, c'est : si la limite est 0, alors ne rien faire ; sinon, imprimer n, puis énumérer les nombres précédents. Si l'on se rend compte que « énumérer les nombres précédents » consiste tout simplement à appeler notre fonction avec la nouvelle limite $n - 1$, on obtient le programme suivant :

```
# let rec compte_à_rebours n =
    if n = 0 then () else
      begin
        print_int n; print_string " ";
        compte_à_rebours (n - 1)
      end;;
compte_à_rebours : int -> unit = <fun>
# compte_à_rebours 10;;
10 9 8 7 6 5 4 3 2 1 - : unit = ()
```

La fonction d'impression des entiers au terminal se nomme tout naturellement `print_int`, par analogie avec la fonction d'impression des chaînes `print_string`. Vous devinez le schéma de nommage de ces fonctions, qui consiste à suffixer le nom `print_` par le type de l'argument. Ainsi, la fonction d'impression des nombres flottants (les valeurs du type `float` que nous verrons au chapitre 8) s'appelle `print_float`, et celle pour les caractères (type `char`) s'appelle `print_char`.

Telle qu'elle est écrite, la fonction `compte_à_rebours` ne termine pas si on lui passe un argument négatif (la définition n'est donc pas bien fondée). Il serait plus sûr de remplacer le test `n = 0` par le test `n <= 0`.

Par curiosité, inversons l'appel récursif et les impressions : autrement dit remplaçons `print_int n; print_string " "; compte_à_rebours (n - 1)` par `compte_à_rebours (n - 1); print_int n; print_string " "`. Et afin de ne pas perdre notre fonction précédente, nous nommons `compte` cette version modifiée de `compte_à_rebours`. Nous obtenons :

```
# let rec compte n =
    if n = 0 then () else
      begin
        compte (n - 1);
        print_int n; print_string " "
      end;;
compte : int -> unit = <fun>
# compte 10;;
1 2 3 4 5 6 7 8 9 10 - : unit = ()
```

Cette fonction compte à l'endroit ! C'est plus difficile à comprendre : l'impression se produit *au retour* des appels récursifs. On doit d'abord atteindre $n = 0$ avant d'imprimer le premier nombre, qui est alors 1. En effet, c'est toujours pendant l'évaluation de `compte 1` que nous appelons `compte 0`. Après avoir imprimé `1`, `compte 1` retourne à l'évaluation de `compte 2`, qui écrit `2` et retourne à `compte 3`, et ainsi de suite.

Nous allons utiliser le mécanisme de « trace » de Caml pour suivre les appels récursifs et les impressions. Ce mécanisme imprime à l'écran les appels successifs d'une fonction, ainsi que les résultats que la fonction calcule. Traçons par exemple la fonction `successeur` :

```
# let successeur x = x + 1;;
successeur : int -> int = <fun>
# trace "successeur";;
La fonction successeur est dorénavant tracée.
- : unit = ()
# successeur 2;;
successeur <-- 2
successeur --> 3
- : int = 3
```

Vous l'avez deviné, l'appel d'une fonction est indiqué par le signe `<--` suivi de l'argument concerné, tandis que le signe `-->` signale un retour de fonction et affiche le résultat obtenu. Nous suivons maintenant le déroulement des appels à `compte` et `compte_à_rebours` :

```
# trace "compte"; trace "compte_à_rebours";;
La fonction compte est dorénavant tracée.
La fonction compte_à_rebours est dorénavant tracée.
- : unit = ()
# compte 3;;
compte <-- 3
compte <-- 2
compte <-- 1
compte <-- 0
compte --> ()
1 compte --> ()
```

```
2 compte --> ()
3 compte --> ()
- : unit = ()
```

On voit clairement que `compte` i s'achève avant l'impression de $i-1$ et que `compte` 0 se termine avant toute impression. Cela contraste avec `compte_à_rebours`, qui imprime i avant l'appel `compte_à_rebours` $(i-1)$:

```
# compte_à_rebours 3;;
compte_à_rebours <-- 3
3 compte_à_rebours <-- 2
2 compte_à_rebours <-- 1
1 compte_à_rebours <-- 0
compte_à_rebours --> ()
compte_à_rebours --> ()
compte_à_rebours --> ()
compte_à_rebours --> ()
- : unit = ()
```

Épeler à l'endroit et à l'envers

Nous allons maintenant montrer la récursion à l'œuvre sur les chaînes de caractères. Pour ce faire, nous avons besoin d'opérations supplémentaires sur les chaînes de caractères. La fonction prédéfinie `string_length` renvoie la longueur d'une chaîne de caractères. La notation s`.[`i`]` désigne le $i^{\text{ième}}$ caractère de la chaîne de caractères s. Le premier caractère a pour numéro 0 ; le dernier a donc pour numéro `string_length` $s-1$.

Accès dans une chaîne s ::= s `.[` *indice* `]`

```
# let le_langage = "Caml";;
le_langage : string = "Caml"
# string_length le_langage;;
- : int = 4
# le_langage.[0];;
- : char = `C`
```

Un caractère en Caml est donc un signe typographique quelconque compris entre deux symboles `‘`.

Voici deux fonctions qui épellent des mots. La première épelle à l'envers, en commençant par le dernier caractère de la chaîne et en s'appelant récursivement sur le caractère précédent.

```
# let rec épelle_envers_aux s i =
    if i >= 0 then
      begin
        print_char s.[i]; print_char ‘ ‘;
        épelle_envers_aux s (i - 1)
      end;;
épelle_envers_aux : string -> int -> unit = <fun>
# let épelle_envers s = épelle_envers_aux s (string_length s - 1);;
épelle_envers : string -> unit = <fun>
```

```
# épelle_envers "snob";;
b o n s - : unit = ()
```

La seconde épelle à l'endroit, en commençant par le premier caractère et en s'appelant récursivement sur le prochain caractère.

```
# let rec épelle_aux s i =
    if i < string_length s then
      begin
        print_char s.[i]; print_char ' ';
        épelle_aux s (i + 1)
      end;;
épelle_aux : string -> int -> unit = <fun>
# let épelle s = épelle_aux s 0;;
épelle : string -> unit = <fun>
# épelle "snob";;
s n o b - : unit = ()
```

Ces deux exemples utilisent une forme nouvelle de l'alternative : la construction « `if` ... `then` ... » sans partie `else`. La partie `else` omise est implicitement complétée par le compilateur, qui ajoute `else ()`, autrement dit « sinon rien ». Ainsi, le code :

```
if i >= 0 then begin ... end;;
```

est compris par Caml comme si nous avions écrit :

```
if i >= 0 then begin ... end else ();;
```

Cette complétion automatique vous explique pourquoi la phrase suivante est mal typée :

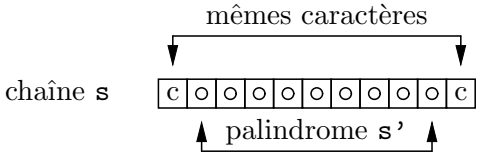
```
# if true then 1;;
Entrée interactive:
>if true then 1;;
>             ^
Cette expression est de type int,
mais est utilisée avec le type unit.
```

Retenons la définition d'une alternative sans partie `else` :

`if` *cond* `then` *e* est équivalent à `if` *cond* `then` *e* `else ()`

Les palindromes

Un palindrome est un mot (ou un groupe de mots sans blancs) qui se lit aussi bien à l'endroit qu'à l'envers. Pour déterminer si un mot constitue un palindrome, il suffit de vérifier que ses premier et dernier caractères sont identiques, puis de vérifier que le mot situé entre ces deux caractères est lui aussi un palindrome.



Il faut aussi remarquer qu'un mot réduit à un seul caractère est un palindrome et que la chaîne vide est elle aussi un palindrome (puisqu'elle se lit aussi bien à l'endroit qu'à

l'envers). Pour obtenir une sous-chaîne d'une chaîne de caractères, on utilise la fonction prédéfinie `sub_string` (sous-chaîne), qui extrait une sous-chaîne partant d'un indice donné et d'une longueur donnée :

```
# sub_string "Le langage Caml" 3 7;;
- : string = "langage"
```

En particulier, `sub_string s 1 (string_length s - 2)` retourne la chaîne `s` privée de son premier et de son dernier caractère. Ceci se traduit par la fonction récursive suivante :

```
# let rec palindrome s =
    let longueur = string_length s in
    if longueur <= 1 then true else
    if s.[0] = s.[longueur - 1]
    then palindrome (sub_string s 1 (longueur - 2))
    else false;;
palindrome : string -> bool = <fun>


# palindrome "serres";;
- : bool = true

# palindrome "toto";;
- : bool = false
```

Opérateurs booléens

Cette fonction s'écrit plus élégamment à l'aide des opérateurs « et » et « ou » des booléens. Par définition, si P et Q sont des booléens, alors l'expression P *et* Q est vraie quand P et Q sont vraies simultanément. Naturellement, l'expression P *ou* Q est vraie dès que P ou bien Q est vraie et *a fortiori* si P et Q sont vraies. En Caml le « ou » se note `||` et le « et » `&&`.

Les opérateurs `||` et `&&` remplacent certaines formes d'expressions conditionnelles. En effet, la construction `if` P `then true else` Q calcule la même valeur booléenne que P `||` Q et de même `if` P `then` Q `else false` calcule la même valeur que P `&&` Q. Bien sûr, `if` P `then true else false` se simplifie en P. On clarifie souvent les programmes en appliquant ces équivalences. Voici donc une version plus simple de `palindrome` qui utilise les opérateurs booléens :

```
# let rec palindrome s =
    let longueur = string_length s in
    (longueur <= 1) ||
      (s.[0] = s.[longueur - 1]) &&
        (palindrome (sub_string s 1 (longueur - 2)));;
palindrome : string -> bool = <fun>
```

Les constructions `||` et `&&` ont les mêmes priorités relatives que `+` et `*`, c'est-à-dire le même parenthésage implicite. Ainsi, de la même façon que `a + b * c` se lit en fait `a + (b * c)`, l'expression `a || b && c` est lue `a || (b && c)` par la machine. On lit alors le code de `palindrome` très naturellement : une chaîne est un palindrome si sa longueur est inférieure à 1, ou si ses caractères de début et de fin sont les mêmes et que la sous-chaîne qu'ils délimitent est un palindrome.

Fonction récursive à plusieurs arguments

Pour plus d'efficacité, nous réécrivons la fonction `palindrome` en comparant directement les caractères de la chaîne argument deux à deux, sans créer de sous-chaînes. On prend donc deux indices dans la chaîne argument s. L'indice i démarre au premier caractère ; l'indice j démarre au dernier caractère (au départ de la boucle on a donc nécessairement $i \leq j$, sauf si la chaîne est vide). À chaque étape, on compare les caractères d'indice i et j. S'ils sont égaux, on continue ; sinon, la chaîne n'est évidemment pas un palindrome. La récursion s'arrête quand l'indice i atteint ou dépasse j. Dans le cas $i = j$, on est sur le caractère central de la chaîne et il n'y a rien à faire (il ne reste qu'un caractère à examiner forcément égal à lui-même) : s est un palindrome. Dans le cas $i > j$, il n'y a rien à faire non plus : on a dépassé le milieu de la chaîne en ayant comparé deux à deux tous les caractères, donc s est un palindrome. Cet exemple nous fournit notre première fonction récursive à plusieurs arguments.

```
# let rec palin s i j =
    (i >= j) || (s.[i] = s.[j]) && (palin s (i + 1) (j - 1));;
palin : string -> int -> int -> bool = <fun>
# let palindrome s = palin s 0 (string_length s - 1);;
palindrome : string -> bool = <fun>
# palindrome "eluparcettecrapule";;
- : bool = true
```

On simplifie encore un peu ce programme en écrivant la `palin` à l'intérieur de `palindrome`, ce qui lui ôte l'argument `s`, qui est lié par la fonction `palindrome` et qui est donc visible par la fonction locale `palin`. C'est la version la plus jolie. Par coquetterie, nous avons aussi supprimé les parenthèses autour des tests, car elles sont implicites.

```
# let palindrome s =
      let rec palin i j =
          i >= j || s.[i] = s.[j] && palin (i + 1) (j - 1) in
      palin 0 (string_length s - 1);;
palindrome : string -> bool = <fun>
# palindrome "tulaStroPécraséCésarcéPortSalut";;
- : bool = true
```

2.2 Définitions par cas : le filtrage

Nous avons donné la définition récursive suivante de la fonction factorielle :

```
let rec factorielle n = if n = 0 then 1 else n * factorielle (n - 1);;
```

Caml dispose d'une manière encore plus concise de définir cette fonction : l'analyse de cas. Il y a ici deux cas possibles pour l'argument de `factorielle`, ou bien c'est 0 ou bien il est différent de 0. On l'écrit ainsi :

```
# let rec factorielle = function
    | 0 -> 1
    | n -> n * factorielle (n - 1);;
factorielle : int -> int = <fun>
```

L'analyse de cas `| 0 -> 1 | n -> n * factorielle (n - 1)` signifie simplement : si l'argument de la fonction est `0` alors renvoyer `1`, sinon nommer `n` l'argument de la fonction et retourner `n * factorielle (n - 1)`. La barre verticale « `|` » introduit donc les cas et correspond à un « ou bien », tandis que la flèche « `->` » indique ce qu'il faut calculer dans le cas correspondant.

L'analyse de cas porte le nom technique de *filtrage* que nous emploierons désormais. Le filtrage est un trait extrêmement puissant de Caml. Il est intégré dans de nombreuses constructions du langage et très fréquemment employé dans les programmes.

Il arrive dans certains cas qu'on n'utilise pas l'argument de la fonction pour calculer le résultat :

```
# let égal_un = function | 1 -> true | x -> false;;
égal_un : int -> bool = <fun>
```

Pour bien montrer que le nom `x` ne sert à rien, puisque sa valeur n'est pas nécessaire pour retourner `false`, on se sert d'un symbole spécial « `_` » (le souligné), qui signifie « dans tous les autres cas » :

```
# let est_un = function | 1 -> true | _ -> false;;
est_un : int -> bool = <fun>

# est_un 1;;
- : bool = true

# est_un 0;;
- : bool = false
```

Nous abordons maintenant un problème apparemment très difficile, qu'une fonction récursive résout sans difficulté et avec une grande élégance.

2.3 Les tours de Hanoi

La légende

Le jeu des tours de Hanoi consiste en une plaquette de bois sur laquelle sont plantées trois tiges. Sur ces tiges sont enfilés des disques de bois dont les diamètres sont tous différents. La seule règle du jeu est de ne jamais poser un disque sur un disque plus petit que lui, et de ne déplacer qu'un seul disque à la fois. Au début du jeu tous les disques sont posés sur la tige de gauche. Le but du jeu est de déplacer les disques d'une tige sur l'autre, sans jamais violer la règle, pour finalement les amener tous sur la tige de droite.

Le jeu original était accompagné d'une notice racontant la légende de moines d'un temple de Hanoi qui passaient leur temps à résoudre ce jeu pour atteindre le nirvana. En effet, les moines croyaient que la fin du monde arriverait lorsque le jeu serait achevé. Leur jeu grandeur nature occupait la cour d'un temple. Il se composait de 64 disques d'or et de trois tiges d'ivoire d'un mètre de haut. Cette légende a été inventée par le mathématicien français Edouard Lucas en 1883.

Notre but est d'écrire un programme qui indique les mouvements à faire pour résoudre le jeu. Si vous n'êtes pas trop impatient, cherchez quelques instants vous-même la solution. C'est difficile, n'est-ce pas ? Et pourtant, ce jeu est étrangement facile à résoudre avec une procédure récursive.

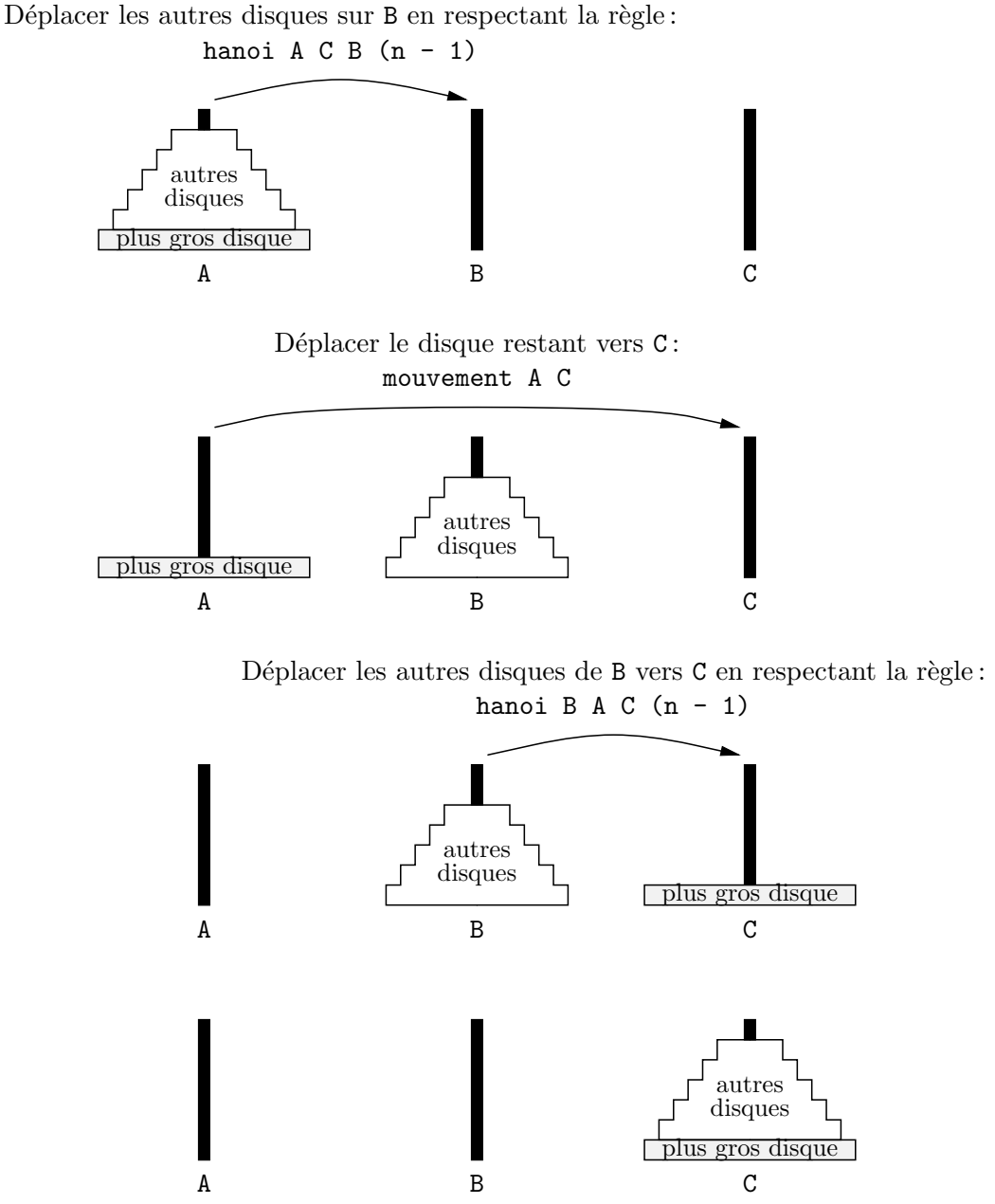


Figure 2.1: Comment résoudre le problème des tours de Hanoi.

Le programme

Supposons que les tiges s'appellent `A`, `B` et `C`, que n soit le nombre de disques, tous posés au départ sur la tige `A`, et que nous devions les mettre sur la tige `C`. L'astuce consiste à se rendre compte que si nous savions comment résoudre le problème pour $n-1$ disques alors nous saurions le faire pour n, sans violer la règle. En effet, si l'on suppose les $n-1$ disques déjà posés sur la tige `B`, le dernier disque encore posé sur la tige `A` est le plus gros disque. Il nous suffit donc de le poser sur la tige `C` qui est vide (pas de violation possible de la règle), puis de déplacer les $n-1$ disques de la tige `B` à la tige `C`. C'est possible puisque nous supposons savoir comment déplacer $n-1$ disques d'une tige à une autre et puisque c'est le plus gros disque qui est maintenant posé sur `C`, il n'y a pas de violation de la règle en posant les $n-1$ disques de `B` sur la tige `C` (voir la figure 2.1). Mais nous savons aussi résoudre le problème pour 0 disques : il n'y a rien à faire. Nous savons donc résoudre le problème des tours de Hanoi pour tout n. C'est encore plus facile à dire en Caml : nous définissons d'abord une fonction auxiliaire pour imprimer les mouvements, puis la procédure principale.

```
# let mouvement de vers =
    print_string
      ("Déplace un disque de la tige " ^ de ^ " à la tige " ^ vers);
    print_newline ();;
mouvement : string -> string -> unit = <fun>

# let rec hanoi départ milieu arrivée = function
    | 0 -> ()
    | n -> hanoi départ arrivée milieu (n - 1);
           mouvement départ arrivée;
           hanoi milieu départ arrivée (n - 1);;
hanoi : string -> string -> string -> int -> unit = <fun>
```

Les noms des arguments `départ`, `milieu` et `arrivée` sont échangés lorsque nous voulons déplacer les disques d'une tige à l'autre : par exemple, pour déplacer un disque de la tige de nom `départ` vers la tige argument `milieu`, nous écrivons `hanoi départ arrivée milieu 1`.

```
# hanoi "A" "B" "C" 3;;
Déplace un disque de la tige A à la tige C
Déplace un disque de la tige A à la tige B
Déplace un disque de la tige C à la tige B
Déplace un disque de la tige A à la tige C
Déplace un disque de la tige B à la tige A
Déplace un disque de la tige B à la tige C
Déplace un disque de la tige A à la tige C
- : unit = ()
```

Vérifiez : le casse-tête est résolu. C'est magique ! On n'a pas vraiment l'impression d'avoir écrit un programme si savant ...

Ne vous inquiétez pas si vous éprouvez des difficultés à comprendre la procédure `hanoi`. C'est normal, car c'est le premier exemple de programme Caml qui nous oblige à changer franchement notre façon d'appréhender les programmes. En effet, il est extrêmement difficile de comprendre *comment* la procédure marche. Au contraire, il

faut se demander *pourquoi* elle marche. Le pourquoi est simple : il est entièrement contenu dans la figure 2.1. Si vous êtes persuadé du bien-fondé de la méthode de résolution que la figure suggère et que vous êtes convaincu que la procédure `hanoi` implémente correctement cette méthode, alors ne cherchez pas plus loin : vous avez tout compris. Si en revanche vous essayez de suivre le déroulement des appels récursifs et les permutations d'arguments qui se déroulent à l'exécution de la procédure (par exemple en utilisant la trace de Caml), vous serez vite perdu. En fait, même si vous suiviez précautionneusement ce déroulement vous n'en apprendriez pas plus, si ce n'est que ça marche, puisque vous constateriez que les bons arguments se mettent en place au bon moment pour produire les bons résultats, comme par miracle. Il faut se décider à penser que ce suivi pas à pas du déroulement des programmes est du ressort de la machine exclusivement. Notre compréhension est de bien plus haut niveau : elle consiste essentiellement à prouver que le programme *ne peut que marcher* ; comment le programme parvient effectivement au bon résultat ne nous regarde pas. Il est heureux que cette noble activité de réflexion sur le bien-fondé d'une méthode de résolution d'un problème nous appartienne en propre, alors que nous déléguons aux machines la mise en œuvre effective. Un équilibre se crée ainsi : si la réflexion sur la méthode est hors de portée de la machine, la gestion sans erreurs des passages de paramètres et la reprise des appels récursifs en suspens est un jeu d'enfant pour la machine, alors que nous serions incapables d'une telle rigueur. Rendons donc aux machines ce qui appartient aux machines.

Pour ceux que cela intéresse, la section suivante esquisse les fondements théoriques de la méthode qui explique pourquoi la procédure `hanoi` marche effectivement. Accessoirement, il permet aussi de calculer la date à laquelle les moines auront achevé leur travail et donne donc une bonne idée de la date de la fin du monde ! Si tout cela ne vous préoccupe pas, passez directement au chapitre suivant.

2.4 Notions de complexité

La complexité est l'étude du nombre d'opérations nécessaires à l'achèvement d'un calcul. Une analyse de complexité permet donc de se faire une idée du temps de calcul nécessaire à l'achèvement d'un programme, en fonction de l'argument qui lui est soumis. En général, on compte le nombre d'opérations élémentaires (additions, multiplications, soustractions et divisions, comparaisons de valeurs, affectations d'éléments de tableau) et/ou le nombre d'appels de fonctions. Par exemple, la fonction `successeur` demande une seule opération, quel que soit son argument. En revanche, la complexité de la fonction `factorielle` dépend de son argument : elle demande n opérations pour l'argument n. Plus précisément, il faut n multiplications, $n+1$ appels récursifs à la fonction `factorielle` et n soustractions. Si l'on considère que ces trois types d'opérations ont des coûts voisins, alors la complexité de `factorielle` est de l'ordre de $2n+(n+1)$, c'est-à-dire de l'ordre de $3n$. On considérera donc que la fonction `factorielle` a une complexité qui augmente au même rythme que son argument, ce qu'on note $O(n)$ et qu'on prononce « grand-o de n ». Plus précisément, $O(n)$ signifie « un certain nombre de fois » n, plus des termes négligeables devant n quand n devient grand, comme par exemple une constante. On ne s'intéresse en effet qu'à un ordre de grandeur de la com-

plexité : cette complexité augmente-t-elle comme l'argument (algorithme *linéaire*), ou comme le carré de l'argument (algorithme *quadratique*), ou comme une exponentielle de l'argument (algorithme *exponentiel*) ? Dans le cas de `factorielle`, on résume l'étude en notant une complexité linéaire $O(n)$, puisque la complexité réelle est $3n + 1$.

Principe de récurrence

Les études de complexité et les définitions récursives de fonctions reposent sur un raisonnement simple sur les propriétés qui concernent les nombres entiers : le principe de récurrence. Nous allons l'expliquer, puis l'utiliser pour démontrer des propriétés de la fonction `hanoi`.

Le principe de récurrence s'énonce informellement ainsi : si une certaine propriété sur les nombres entiers est vraie pour 0 et si la propriété est vraie pour le successeur d'un nombre dès qu'elle est vraie pour ce nombre, alors cette propriété est vraie pour tous les nombres. Formellement : soit $P(n)$ une propriété qui dépend d'un entier n. Si les phrases suivantes sont vraies :

1. $P(0)$ est vraie,
2. si $P(n)$ est vraie alors $P(n + 1)$ est vraie,

alors $P(n)$ est vraie pour tout n.

Ce principe est en fait évident : les deux propriétés demandées par le principe de récurrence permettent facilement de démontrer la propriété P pour toute valeur entière. Par exemple, supposons que P vérifie les deux propriétés et qu'on veuille démontrer que P est vraie pour 2. Puisque P est vraie pour 0 elle est vraie pour son successeur, 1. Mais puisque P est vraie pour 1 elle est vraie pour son successeur, donc elle est vraie pour 2. Il est clair que ce raisonnement se poursuit sans problème pour tout nombre entier fixé à l'avance.

C'est ce principe que nous avons utilisé pour résoudre le problème des tours de Hanoi :

1. nous avons montré que nous savions le résoudre pour 0 disque ;
2. nous avons montré qu'en sachant le résoudre pour $n - 1$ disques nous savions le résoudre pour n disques.

Ces deux cas correspondent exactement aux deux clauses de la fonction `hanoi` (cas `0 ->` et cas `n ->`). Le principe de récurrence nous prouve donc que nous savons effectivement résoudre le problème pour tout n, même si cela ne nous apparaissait pas clairement au départ.

La difficulté intuitive de ce genre de définitions récursives est d'oser utiliser l'hypothèse de récurrence : il faut supposer qu'on sait déjà faire pour $n - 1$ disques et écrire le programme qui résout le problème pour n disques. Dans la procédure `hanoi`, on suppose ainsi deux fois que la fonction saura bien faire toute seule pour $n - 1$ disques et l'on ne s'occupe que de déplacer le gros disque, ce qui semble un travail facile. Finalement, on a l'impression de voir tourner du code que l'on n'a pas écrit, tellement il semble astucieux à l'exécution.

L'écriture de fonctions récursives se réduit ainsi très souvent au schéma :

```
let rec f = function
  | 0 ->  «solution simple»
  | n -> ... f (n - 1) ... f (n - 1) ...;;
```

On démontre en mathématiques qu'il n'est pas interdit d'appeler `f` sur d'autres arguments que `n - 1`, pourvu qu'ils soient plus petits que `n` (par exemple `n - 2`), mais alors il faut prévoir d'autres cas simples (par exemple `1 ->`). Un exemple de ce schéma de programme est la fonction de Fibonacci définie par :

```
# let rec fib = function
    | 0 -> 1
    | 1 -> 1
    | n -> fib (n - 1) + fib (n - 2);;
fib : int -> int = <fun>
# fib 10;;
- : int = 89
```

Remarquez que cette fonction fait effectivement deux appels récursifs sur deux valeurs différentes, mais toutes les deux plus petites que l'argument donné.

Complexité de la procédure `hanoi`

Il est facile d'écrire un programme qui compte le nombre de mouvements nécessaires pour résoudre le jeu pour n disques : il y a 0 mouvement à faire pour 0 disque, l'appel à la procédure `mouvement` produit 1 mouvement et le nombre de mouvements nécessaires aux appels récursifs est forcément compté par la fonction récursive de comptage que nous sommes en train de définir. En effet, on suppose une fois de plus que pour $n-1$ la fonction « sait faire » et on se contente de trouver le résultat pour n.

```
# let rec compte_hanoi départ milieu arrivée = function
    | 0 -> 0
    | n -> compte_hanoi départ arrivée milieu (n - 1) + 1 +
           compte_hanoi milieu départ arrivée (n - 1);;
compte_hanoi : 'a -> 'a -> 'a -> int -> int = <fun>
```

Les arguments contenant les noms des tiges sont bien sûr inutiles et il suffit d'écrire :

```
# let rec compte_hanoi_naïf = function
    | 0 -> 0
    | n -> compte_hanoi_naïf (n - 1) + 1 + compte_hanoi_naïf (n - 1);;
compte_hanoi_naïf : int -> int = <fun>
```

qu'on simplifie encore en

```
# let rec compte_hanoi = function
    | 0 -> 0
    | n -> (2 * compte_hanoi (n - 1)) + 1;;
compte_hanoi : int -> int = <fun>


# compte_hanoi 3;;          # compte_hanoi 10;;         # compte_hanoi 16;;
- : int = 7                 - : int = 1023             - : int = 65535
```

On devine la propriété suivante : pour tout n, `compte_hanoi` $(n) = 2^n - 1$. Nous allons la démontrer en utilisant le principe de récurrence. Nous définissons donc formellement

la propriété P par : $P(n)$ est vraie si et seulement si `compte_hanoi` $(n) = 2^n - 1$. La proposition $P(0)$ est vraie car `compte_hanoi` $(0) = 0$ et $2^0 - 1 = 1 - 1 = 0$. Supposons $P(n)$ vraie et montrons qu'alors $P(n+1)$ est vraie. Pour montrer $P(n+1)$, il faut démontrer

$$\texttt{compte_hanoi}\ (n+1) = 2^{n+1} - 1.$$

Or, d'après la définition de la fonction `compte_hanoi`, on a :

$$\texttt{compte_hanoi}\ (n+1) = 2 \times \texttt{compte_hanoi}\ ((n+1) - 1) + 1,$$

soit `compte_hanoi` $(n+1) = 2 \times$ `compte_hanoi` $(n) + 1$. Mais, par hypothèse de récurrence, $P(n)$ est vraie, donc `compte_hanoi` $(n) = 2^n - 1$. En reportant dans l'égalité précédente, on obtient :

$$\texttt{compte_hanoi}\ (n+1) = 2 \times (2^n - 1) + 1.$$

Mais $2 \times (2^n - 1) + 1 = 2^{n+1} - 2 + 1 = 2^{n+1} - 1$, donc

$$\texttt{compte_hanoi}\ (n+1) = 2^{n+1} - 1$$

et $P(n+1)$ est vraie. Il s'ensuit, d'après le principe de récurrence, que $P(n)$ est vraie pour tout n.

Avec ce nouveau résultat, nous sommes autorisés à redéfinir `compte_hanoi` comme la fonction qui à n associe $2^n - 1$. Pour avoir une idée du nombre de mouvements nécessaires pour résoudre le problème avec 64 disques, nous sommes obligés de faire les calculs en « virgule flottante » car le résultat excède de beaucoup la limite supérieure des entiers représentables en Caml. Nous reviendrons plus tard sur les nombres en virgule flottante, aussi appelés nombres flottants (chapitre 8). Pour l'instant il suffit de savoir qu'un nombre flottant est caractérisé par le point qui précède sa partie décimale et que les opérations associées aux flottants sont suffixées également par un point (`+.`, `-.`, `*.`, etc.). Nous implémentons donc notre fonction en utilisant la fonction « puissance » des nombres flottants (`power`).

```
# let compte_hanoi_rapide n = power 2.0 n -. 1.0;;
compte_hanoi_rapide : float -> float = <fun>
# compte_hanoi_rapide 64.0;;
- : float = 1.84467440737e+19
```

Un algorithme correct mais inutilisable

Grâce à notre démonstration mathématique, nous avons établi une formule de calcul direct du nombre de mouvements nécessaires à la résolution du jeu pour n disques. Nous avons ainsi très fortement accéléré la fonction `compte_hanoi`. C'était indispensable car notre première version, la fonction `compte_hanoi_naïf`, quoique parfaitement correcte d'un point de vue mathématique, n'aurait pas pu nous fournir le résultat pour 64. En effet cette version calcule son résultat en utilisant uniquement l'addition. Plus précisément, elle n'ajoute toujours que des 1 : il lui aurait donc fallu faire $2^{64} - 1$ additions. Même en supposant qu'on fasse 1 milliard d'additions par seconde, ce qui est à la limite de la technologie actuelle, il aurait fallu, avec le programme de la première version de `compte_hanoi`,

```
# let nombre_de_secondes_par_an = 3600.0 *. 24.0 *. 365.25;;
nombre_de_secondes_par_an : float = 31557600.0
# let nombre_d'additions_par_an = nombre_de_secondes_par_an *. 1E9;;
nombre_d'additions_par_an : float = 3.15576e+16
# compte_hanoi_rapide 64.0 /. nombre_d'additions_par_an;;
- : float = 584.542046091
```

c'est-à-dire plus de 584 années pour achever le calcul ! Nous sommes donc ici en présence d'une fonction qui donne effectivement le bon résultat au sens des mathématiques, mais qui le calcule tellement lentement qu'elle devient inutilisable. À la différence des mathématiques, il ne suffit donc pas en informatique d'écrire des programmes corrects, il faut encore que leur complexité ne soit pas trop élevée pour qu'ils calculent le résultat correct en un temps raisonnable.

La fonction `compte_hanoi_naïve` nécessite $2^n - 1$ additions pour l'argument n. Son temps de calcul est donc proportionnel à une puissance (2^n) dont l'exposant est son argument n : l'algorithme est exponentiel. La seconde version utilisant la multiplication nécessite n multiplications, l'algorithme est donc linéaire. Un algorithme linéaire demande un temps de calcul qui augmente comme la valeur de son argument ($O(n)$), ce qui est raisonnable. En effet, cette version nous aurait permis d'obtenir notre résultat, puisque pour $n = 64$ il aurait fallu 64 multiplications seulement. La dernière version, quant à elle, est *en temps constant*. Elle ne nécessite que deux opérations flottantes quel que soit son argument : c'est l'algorithme idéal. On retiendra qu'un algorithme exponentiel est vite susceptible d'exiger un temps de calcul prohibitif quand son argument augmente.

Date de la fin du monde

Calculons le nombre d'années nécessaires aux moines pour achever leur jeu à 64 disques. Supposons qu'ils puissent effectuer sans arrêt, jour et nuit, dix mouvements par secondes, ce qui est vraiment le maximum qu'on puisse exiger de ces pauvres moines. Il leur faudrait alors :

```
# let nombre_de_mouvements_par_an =
      nombre_de_secondes_par_an *. 10.0;;
nombre_de_mouvements_par_an : float = 315576000.0
# compte_hanoi_rapide 64.0 /. nombre_de_mouvements_par_an;;
- : float = 58454204609.1
```

soit plus de 58 milliards d'années. C'est beaucoup plus que la durée de vie estimée du Soleil. Il semble donc que l'heure de la fin du monde aura sonné très longtemps avant la fin du jeu !

Calcul de la complexité de la seconde version

Dans la section précédente, nous avons affirmé que la seconde version de `compte_hanoi` :

```
# let rec compte_hanoi = function
    | 0 -> 0
    | n -> 2 * compte_hanoi (n - 1) + 1;;
compte_hanoi : int -> int = <fun>
```

nécessitait n multiplications. La démonstration en est très simple. Nous noterons $Op($`compte_hanoi` $(n))$ le nombre d'opérations nécessaires pour effectuer le calcul de `compte_hanoi` (n) à l'aide de cette version de `compte_hanoi`. Nous démontrons par récurrence la propriété $P(n)$ définie par : $P(n)$ est vraie si et seulement si $Op($`compte_hanoi` $(n)) = n$. La propriété $P(0)$ est vraie car $Op($`compte_hanoi` $(0)) = 0$. Supposons $P(n)$ vraie et montrons qu'alors $P(n+1)$ est vraie. Pour montrer $P(n+1)$, il faut démontrer $Op($`compte_hanoi` $(n+1)) = (n+1)$. Or, d'après le code de la fonction `compte_hanoi`, quand on a le résultat de `compte_hanoi (n - 1)`, il faut faire une multiplication de plus pour obtenir `compte_hanoi (n)`. On a donc : $Op($`compte_hanoi` $(n+1)) = 1 + Op($`compte_hanoi` $(n))$; mais, d'après l'hypothèse de récurrence, $Op($`compte_hanoi` $(n)) = n$, et donc $Op($`compte_hanoi` $(n+1)) = n+1$. Il s'ensuit que $P(n)$ est vraie pour tout n.

Remarquons pour finir que nous avons calculé la complexité de `hanoi` en utilisant la fonction `compte_hanoi`, dont nous avons dû à nouveau étudier la complexité, pour l'optimiser (sous peine de ne pas obtenir effectivement la complexité de `hanoi`). Il faut décidément réfléchir sur les programmes qu'on écrit ...

3
Programmation impérative

Où l'on apprend que $2x + 2x$ *font* $4x$.

Nous mettons en place dans ce chapitre quelques outils indispensables à la programmation impérative. En particulier, nous introduisons la notion de *tableau*, et l'utilisons pour calculer des identités remarquables. Nous serons par exemple en mesure d'établir par programme la formule $(x + 1)^2 = x^2 + 2x + 1$. En termes savants nous ferons du calcul formel sur des polynômes à une indéterminée. Si vous savez déjà qu'il y a autre chose dans la vie que la programmation fonctionnelle et que vous connaissez les boucles « for » et « while », vous pouvez sauter ce chapitre.

3.1 La programmation impérative

Jusqu'à présent, nous avons écrit de petits programmes dans un sous-ensemble de Caml : la partie *déclarative*, la plus proche des mathématiques. Nous avons toujours défini des fonctions qui retournaient le résultat que nous voulions calculer. Ces fonctions *calculent* le résultat souhaité au sens des calculs mathématiques, c'est-à-dire par simplifications successives d'une expression. Ce style de programmation à l'aide de fonctions s'appelle la *programmation fonctionnelle*.

Une autre façon de calculer consiste à considérer qu'un calcul est un processus évolutif, où le temps a son importance. Il s'agit de *modifier un état* : l'ordinateur commence l'exécution du programme dans un certain état initial, que l'exécution du programme modifie jusqu'à parvenir à un état final qui contient le résultat voulu. On change l'état courant par modification du contenu de la mémoire de l'ordinateur (à l'aide d'*affectations*), ou encore par interaction avec le monde extérieur : interrogation de l'utilisateur, affichage de résultats, lecture ou écriture de fichiers, bref tout ce qu'on nomme les entrées-sorties. Toutes ces opérations qui modifient physiquement le contenu des adresses mémoire sont appelées *effets* (ou encore effets de bord) :

> Un *effet* est une modification d'une case de la mémoire (tableau ou référence), ou encore une interaction avec le monde extérieur (impression ou lecture).

Ce style de programmation par effets s'appelle la *programmation impérative*. Ce nom provient évidemment de la signification du mode impératif dans la conjugaison des verbes. En effet, les programmes impératifs décrivent explicitement à la machine la suite des opérations à effectuer (fais ci, fais ça). Au contraire, en programmation fonctionnelle, on laisse la machine calculer le résultat comme elle le peut à partir d'une formule, sans lui préciser complètement l'ordre dans lequel elle doit opérer. Par exemple, pour calculer le carré d'un nombre `x`, on écrit `x * x` en programmation fonctionnelle. Au contraire, une méthode impérative serait de réserver une case mémoire comme accumulateur, de l'initialiser avec `x`, puis de remplacer le contenu de cet accumulateur par son contenu multiplié par lui-même. Le résultat cherché serait maintenant dans l'accumulateur. Dans un cas si simple, ces descriptions sont évidemment caricaturales, mais l'idée est la bonne.

Le style impératif implique la modification de l'état de la mémoire, donc l'utilisation de structures de données modifiables (par exemple les tableaux dont les éléments peuvent être changés dynamiquement) et l'emploi de *commandes*. Les commandes sont des expressions qui ne retournent pas de valeurs intéressantes ; leur résultat est simplement une modification de l'état courant, c'est-à-dire un effet. Lorsqu'une fonction se contente d'exécuter une série de commandes, on l'appelle souvent *procédure*. Une procédure en Caml est donc simplement une fonction qui se contente de faire des effets, sans produire de résultat au sens mathématique.

Nous aurions pu nous cantonner au sous-ensemble fonctionnel de Caml et cependant écrire de très jolis programmes. Mais c'eût été donner une fausse image de Caml : ce qui fait la puissance du langage c'est justement qu'il ne se limite pas à la programmation fonctionnelle, mais intègre harmonieusement programmation fonctionnelle et programmation impérative. De plus, nous cherchons avant tout à vous montrer les programmes les plus simples et les plus clairs possibles : nous avons donc besoin de tous les outils que Caml met à notre disposition.

De surcroît, la programmation impérative n'est pas seulement indispensable pour traiter les problèmes d'interaction avec le monde extérieur (entrées-sorties). Dans certains cas un algorithme, c'est-à-dire une méthode de résolution d'un problème, exige moins de calculs lorsqu'il est écrit en style impératif que lorsqu'il est écrit en style fonctionnel. Enfin, certains algorithmes s'expriment naturellement en termes d'évolution d'un état ; la programmation impérative s'impose alors.

Nous avons pour l'instant illustré les effets d'entrées-sorties, plus précisément les impressions. Nous allons maintenant faire des effets sur la mémoire, ce qu'on appelle aussi des *modifications physiques* ou *modifications en place* de données. Pour cela il nous faut disposer de cases mémoire modifiables par le programme. Caml propose pour cela les notions de *références* et de *tableaux*. Nous commençons par étudier les tableaux, qui sont plus simples.

Puisque la notion de temps intervient en programmation impérative, il nous faut un moyen de spécifier au langage « fait ceci d'abord » et « fait cela ensuite » : c'est la notion de *séquence* que nous avons déjà vue au chapitre 1. Nous avons également besoin de répéter des suites d'effets : c'est la notion de *boucles*. Nous décrivons ces construction, puis appliquons ces outils au calcul sur les polynômes.

3.2 Boucles

Caml fournit deux sortes de boucles pour répéter des effets : la boucle « pour » et la boucle « tant que ». La boucle « pour » répète un calcul un nombre de fois fixé à l'avance ; la boucle « tant que » répète un calcul tant qu'une condition reste vraie.

Boucle « tant que »

Boucle « *tant que* »	::=	`while` *expression*	(while : tant que)
		`do` *expression* `done`	(do : faire, done : fait)

La signification de `while` *condition* `do` *actions* `done` est simplement de faire les *actions* tant que la *condition* est vraie. La condition est testée au début de chaque itération. Si elle est initialement fausse, les actions ne sont jamais exécutées. Dans certains cas, la boucle « tant que » sert à répéter indéfiniment les mêmes actions jusqu'à un événement exceptionnel. Dans ce cas, la condition de boucle est tout simplement le booléen `true`, comme dans `while true do` *actions* `done`.

Boucle « pour »

Boucle « *pour* »	::=	`for` *ident* `=` *expression*	(for : pour)
		(`to` | `downto`) *expression*	(to : jusqu'à, down : en bas)
		`do` *expression* `done`	(do : faire, done : fait)

La *sémantique*, c'est-à-dire la signification, de l'expression `for i =` *début* `to` *fin* `do` *actions* `done` est de faire les *actions* avec `i` = *début*, puis avec `i` = *début* + 1 et ainsi de suite, jusqu'à `i` = *fin*. En particulier, si *début* > *fin*, on n'évalue jamais *actions*. Pour la version `downto`, on décrémente l'indice de boucle `i` (on lui soustrait 1) à chaque tour, au lieu de l'incrémenter (lui ajouter 1). L'indice de boucle est forcément du type entier. Le nom associé à l'indice de boucle est introduit par la boucle (comme par une liaison `let`) ; sa liaison n'est valide que pendant le corps de la boucle. Prenons un exemple simple : nous imprimons les dix chiffres à l'aide d'une boucle de 0 à 9. Nous définissons une procédure `imprime_chiffre` dont l'argument est « rien », et nous la déclenchons en l'appliquant à « rien ».

```
# let imprime_chiffres () =
      for i = 0 to 9 do
        print_int i
      done;
      print_newline ();;
imprime_chiffres : unit -> unit = <fun>
# imprime_chiffres ();;
0123456789
- : unit = ()
```

3.3 Manipulation de polynômes

Nous continuons l'apprentissage de la programmation impérative par l'étude des *tableaux*. À titre d'illustration, nous écrivons un jeu de fonctions qui implémentent les opérations de base sur les polynômes. Avant de nous lancer dans la programmation, nous rappelons brièvement ce que sont les polynômes.

Les polynômes à une indéterminée

Des classes élémentaires, on retient souvent qu'un polynôme est une somme de puissances de x. Par exemple, $p = x^2+2x+3$ est un polynôme. La variable x est appelée l'indéterminée du polynôme. Un polynôme est une somme de termes élémentaires qu'on nomme monômes (par exemple x^2 et $2x$). Étant donnée une variable x, on appelle monôme de coefficient a_i et de degré i l'expression $a_i\,x^i$. Le degré d'un polynôme est celui de son monôme de plus haut degré. On rappelle que $x^1 = x$ et $x^0 = 1$. Le monôme de degré 0 est donc réduit à une constante (c'est 3 pour p) et celui de degré 1 au produit d'un nombre par l'indéterminée (c'est $2x$ pour p). D'autre part, nous utiliserons la propriété : pour tout n et m entiers positifs, $x^n \times x^m = x^{n+m}$.

Nous modélisons les polynômes à l'aide de tableaux d'entiers : le tableau des coefficients de leurs monômes. Les degrés seront donc implicites, simplement déterminés par l'indice du coefficient dans le tableau qui représente le polynôme. Par exemple, le polynôme $p = x^2 + 2x + 3$ sera représenté par le tableau contenant les nombres 3, 2, 1 dans cet ordre, puisque 3 est le coefficient de degré 0 de p, 2 est le coefficient de degré 1 et 1 le coefficient de degré 2. Nous étudions donc maintenant brièvement les tableaux de Caml.

Tableaux

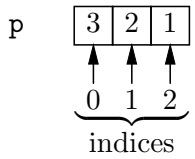
Les tableaux, aussi appelés « vecteurs », sont des suites finies et modifiables de valeurs d'un même type. Leur type est noté *'a vect* (où *'a* signifie « n'importe quel type »). Puisque les éléments des tableaux sont tous de la même nature (du même type), on qualifie les tableaux de suites *homogènes* de valeurs. Les valeurs d'un tableau sont enregistrées dans des cellules de mémoire consécutives. Les positions des éléments dans un tableau débutent à la position 0.

Construction de tableaux

Un tableau se définit de deux façons : soit en dressant directement la liste de ses éléments, soit en créant le tableau et en remplissant ses cases ultérieurement. Si un tableau est défini par la liste de ses éléments, cette liste est entourées des symboles `[|` et `|]`, tandis que les éléments sont séparés par des « `;` ». Notre polynôme $p = x^2+2x+3$ se définit donc par la phrase :

```
# let p = [| 3; 2; 1 |];;
p : int vect = [|3; 2; 1|]
```

Graphiquement, on représente naturellement les tableaux par une succession de cases. Par exemple, `p` sera représenté ainsi :



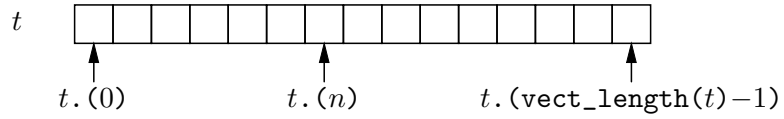
Pour construire des tableaux dont on remplira les cases plus tard, on dispose de la fonction prédéfinie `make_vect`. Avec cette fonction, on crée un tableau en donnant sa taille et un élément qui sera mis au départ dans toutes les cases du tableau : la valeur d'initialisation du tableau. Définissons par exemple un tableau de taille 4 contenant des 2 et un tableau de taille 3 contenant la chaîne `"Bonjour"` :

```
# let q = make_vect 4 2;;
q : int vect = [|2; 2; 2; 2|]
# let r = make_vect 3 "Bonjour";;
r : string vect = [|"Bonjour"; "Bonjour"; "Bonjour"|]
```

La taille d'un tableau s'obtient en appelant la primitive `vect_length`.

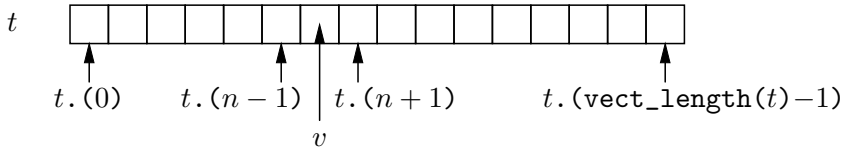
```
# vect_length q;;
- : int = 4
```

Une fois le tableau créé, on peut consulter et modifier le contenu de ses cases. Si t est un tableau et n un entier, t`.(`n`)` désigne l'élément d'indice n du tableau t.




```
# let a0 = p.(0);;
a0 : int = 3
```

On affecte la valeur v à la case n du tableau t par la construction t`.(`n`) <-` v. Cela correspond graphiquement à :



La valeur retournée par cette construction est `()`, la valeur « rien ».

```
# q.(0) <- 1;;
- : unit = ()
# q;;
- : int vect = [|1; 2; 2; 2|]
# r.(1) <- "tout"; r.(2) <- "le monde!";;
- : unit = ()
# r;;
- : string vect = [|"Bonjour"; "tout"; "le monde!"|]
```

Nous savons maintenant définir des tableaux, en lire et modifier les éléments. Il nous faut encore apprendre à les parcourir. C'est très facile en utilisant les boucles que nous avons décrites à la section précédente. Puisqu'il s'agit de parcourir un tableau,

on connaît à l'avance le nombre de répétitions : on utilise donc une boucle « pour ». Le parcours complet d'un tableau `t` s'effectue par une boucle commençant en 0 et finissant en `vect_length t - 1`. En effet, puisque les indices d'éléments de tableaux commencent toujours à 0, le dernier élément d'un tableau a pour indice la longueur du tableau *moins un*. Par exemple :

```
# for i = 0 to vect_length r - 1 do
    print_string r.(i)
  done;;
Bonjourtoutle monde!- : unit = ()
```

Pour rendre la sortie plus jolie, il suffit d'ajouter un blanc après chaque élément :

```
# for i = 0 to vect_length r - 1 do
    print_string r.(i);
    print_string " "
  done;;
Bonjour tout le monde! - : unit = ()
```

Syntaxe des tableaux

Pour mémoire, voici la syntaxe BNF correspondant à ces deux constructions et à la définition des tableaux sous la forme de liste d'éléments.

La syntaxe des définitions de tableaux est la suivante :

$$\textit{Tableaux} \quad ::= \quad \texttt{[|}\ \textit{expression}\ (\ \texttt{;}\ \textit{expression})^*\ \texttt{|]}$$

Nous utilisons ici un nouveau symbole pour la description des constructions syntaxiques qui acceptent les répétitions : l'étoile « * ». La formule *quelque-chose** signifie la répétition de *quelque-chose* un nombre quelconque de fois, y compris zéro fois si nécessaire (ce qui correspond alors à ignorer complètement *quelque-chose*). Nous indiquons ainsi que le premier élément du tableau est éventuellement suivi d'autres éléments, en nombre quelconque, séparés par des points-virgules.

La syntaxe de l'affectation et de l'accès aux éléments de tableaux est la suivante :

$$\begin{array}{rcl} \textit{Accès dans un tableau} & ::= & \textit{vect}\ \texttt{.(}\ \textit{indice}\ \texttt{)} \\ \textit{Modification d'un élément de tableau} & ::= & \textit{vect}\ \texttt{.(}\ \textit{indice}\ \texttt{)}\ \texttt{<-}\ \textit{expression} \end{array}$$

Attention à la signification des parenthèses dans cette description. Elles font ici partie de la syntaxe décrite (il faut les écrire dans les programmes), alors que dans la notation (`;` *expression*)*, les parenthèses nous servaient à regrouper les constructions syntaxiques « `;` » et « *expression* ». (La différence de nature des parenthèses se traduit par un changement de police de caractères.)

3.4 Impression des polynômes

Nous savons maintenant représenter les polynômes à l'aide des tableaux. Pour les manipuler, nous savons parcourir leurs coefficients à l'aide d'une boucle `for`. Nous pouvons donc commencer l'implémentation des opérations élémentaires sur les polynômes.

Commençons par écrire une procédure d'impression des polynômes pour visualiser simplement nos résultats. Il suffit de parcourir le tableau représentant le polynôme en imprimant ses monômes. Nous écrivons donc d'abord la fonction d'impression d'un

monôme de coefficient c et de degré d. C'est très simple : si le degré est 0, il suffit d'écrire le coefficient ; sinon, on écrit le coefficient et le degré sous la forme c`X^`d. Par exemple, $3x^2$ sera écrit `3x^2`. Cet imprimeur n'est pas très élaboré : il se contente de ne pas écrire les monômes nuls ni les coefficients égaux à 1 ; il traite aussi spécialement le cas particulier des monômes de degré 0 et 1. Ainsi il écrit `x^2` pour le monôme $1x^2$, `3` pour le monôme $3x^0$ et `4x` pour le monôme $4x^1$.

```
# let imprime_monôme coeff degré =
      if degré = 0 then print_int coeff else
      if coeff <> 0 then
        begin
          print_string " + ";
          if coeff <> 1 then print_int coeff;
          print_string "x";
          if degré <> 1 then
            begin print_string "^"; print_int degré end
        end;;
imprime_monôme : int -> int -> unit = <fun>
```

La primitive `<>` correspond au prédicat mathématique $\neq$ et teste donc si deux valeurs sont différentes.

Il est temps de donner le nom technique des polynômes modélisés par des tableaux d'entiers : on les appelle polynômes *pleins*, ce qui signifie simplement que leur représentation comporte la liste exhaustive de leurs monômes, y compris ceux dont le coefficient est nul. C'est pourquoi nous appelons la fonction d'impression de ces polynômes `imprime_polynôme_plein`. Cette fonction se contente d'itérer la procédure `imprime_monôme` sur tout le polynôme, à l'aide d'une boucle « pour ».

```
# let imprime_polynôme_plein p =
      for i = 0 to vect_length p - 1 do imprime_monôme p.(i) i done;;
imprime_polynôme_plein : int vect -> unit = <fun>
```

Le polynôme $p = x^2 + 2x + 3$ s'imprime comme suit :

```
# imprime_polynôme_plein p;;
3 + 2x + x^2- : unit = ()
```

Addition des polynômes

L'addition des polynômes se fait monôme par monôme, en ajoutant les coefficients des monômes de même degré :

$$\begin{aligned}(1 + 2x + 3x^2) + (4 + 5x + 6x^2) &= (1 + 4) + (2 + 5)x + (3 + 6)x^2 \\ &= 5 + 7x + 9x^2.\end{aligned}$$

Mathématiquement, on a : si $P = p_0 + p_1x + p_2x^2 + \cdots + p_mx^m$ et $Q = q_0 + q_1x + q_2x^2 + \cdots + q_nx^n$, alors

$$P + Q = (p_0 + q_0) + (p_1 + q_1)x + (p_2 + q_2)x^2 + \cdots + (p_{\max(m,n)} + q_{\max(m,n)})x^{\max(m,n)}$$

Cette définition utilise la convention qu'un coefficient de degré supérieur au degré du polynôme est implicitement 0. On peut exprimer ces formules de façon plus synthétique

en introduisant la notation mathématique Σ pour résumer les sommes de termes : si *formule* est une expression qui dépend de l'entier i, on note

$$\sum_{i=0}^{n} \mathit{formule}\,(i) \quad \text{pour} \quad \mathit{formule}\,(0) + \mathit{formule}\,(1) + \cdots + \mathit{formule}\,(n).$$

(La notation $\sum_{i=0}^{n} \mathit{formule}\,(i)$ se lit « somme de i égale 0 à n de *formule* (i) ».) Par exemple, si la formule est réduite à i, on obtient la somme des nombres de 0 à n :

$$\sum_{i=0}^{n} i \;=\; 0 + 1 + \cdots + \; n.$$

De même, si la formule est i^2, la somme correspondante est celle des carrés des nombres entre 0 et n :

$$\sum_{i=0}^{n} i^2 \;=\; 0^2 + 1^2 + \cdots + \; n^2.$$

On exprime ainsi de manière très concise deux polynômes généraux P et Q et leur somme $P + Q$:

$$\text{Si} \quad P = \sum_{i=0}^{m} p_i\, x^i \quad \text{et} \quad Q = \sum_{i=0}^{n} q_i\, x^i \quad \text{alors} \quad P + Q \;=\; \sum_{i=0}^{\max(m,n)} (p_i + q_i)\, x^i.$$

La traduction en Caml de cette définition est très simple : on crée d'abord un tableau `somme`, pour contenir la somme des deux polynômes P et Q. Ce tableau a pour longueur le maximum des longueurs de P et Q, qu'on calcule avec la fonction prédéfinie `max`. Il suffit de recopier les coefficients de l'un des polynômes dans les cases du résultat, puis d'y ajouter les coefficients de l'autre.

```
# let ajoute_polynômes_pleins p q =
    let somme = make_vect (max (vect_length p) (vect_length q)) 0 in
    for i = 0 to vect_length p - 1 do
      somme.(i) <- p.(i)
    done;
    for i = 0 to vect_length q - 1 do
      somme.(i) <- somme.(i) + q.(i)
    done;
    somme;;
ajoute_polynômes_pleins : int vect -> int vect -> int vect = <fun>
```

Comme promis, nous pouvons maintenant calculer la valeur de $2x + 2x$:

```
# imprime_polynôme_plein (ajoute_polynômes_pleins [|0; 2|] [|0; 2|]);;
0 + 4x- : unit = ()
```

et vérifier un de nos calculs précédents :

```
# imprime_polynôme_plein
    (ajoute_polynômes_pleins [|1; 2; 3|] [|4; 5; 6|]);;
5 + 7x + 9x^2- : unit = ()
```

et même (qui l'eût cru ?) calculer $x - x$ et trouver 0 :

```
# imprime_polynôme_plein
    (ajoute_polynômes_pleins [|0; 1|] [|0; -1|]);;
0- : unit = ()
```

Multiplication des polynômes

On définit le produit des polynômes en utilisant les règles classiques de développement des expressions algébriques. En termes savants, on dit que l'on utilise la distributivité de la multiplication par rapport à l'addition. Par exemple, pour tout polynôme Q, on a $(1 + 2x + 3x^2) \times Q = 1 \times Q + 2x \times Q + 3x^2 \times Q$, et donc

$$\begin{aligned}
&(1 + 2x + 3x^2) \times (4 + 5x + 6x^2) \\
&\quad = \quad 1 \times (4 + 5x + 6x^2) + 2x \times (4 + 5x + 6x^2) + 3x^2 \times (4 + 5x + 6x^2) \\
&\quad = \quad (4 + 5x + 6x^2) + (8x + 10x^2 + 12x^3) + (12x^2 + 15x^3 + 18x^4) \\
&\quad = \quad 4 + 13x + 28x^2 + 27x^3 + 18x^4.
\end{aligned}$$

La remarque fondamentale est que le produit des coefficients des monômes de degré i du premier polynôme et de degré j du second forme une partie du coefficient du monôme de degré $i + j$ du produit. Par exemple, pour les monômes $2x$ et $6x^2$, le produit 2×6 entrera dans la composition du coefficient du monôme de degré trois du résultat, ce qui signifie simplement que $2x \times 6x^2 = 12x^3$. Pour prendre en compte le produit des monômes $2x$ et $6x^2$ dans le résultat final, il suffit donc d'ajouter le produit 2×6 dans la case correspondant au coefficient de x^3 du résultat final. Ainsi, notre procédure Caml va parcourir les monômes des deux polynômes deux à deux, en les multipliant et en enregistrant le produit de leurs coefficients dans le monôme de degré correspondant du produit.

Pour les fanatiques des formules, ceci se traduit par la définition mathématique suivante :

$$\text{Le produit des deux polynômes } P = \sum_{i=0}^{m} p_i\, X^i \quad \text{et} \quad Q = \sum_{j=0}^{n} q_j\, X^j$$

$$\text{est le polynôme } (P \times Q) = \sum_{k=0}^{m+n} r_k\, X^k \quad \text{avec} \quad r_k = \sum_{i+j=k} p_i \times q_j,$$

$$\text{ou de façon équivalente, } (P \times Q) = \sum_{k=0}^{m+n} \left(\sum_{i=0}^{k} p_i \times q_{k-i} \right) X^k.$$

Remarquez que ces formules d'apparence rébarbative ne font que résumer de façon très succinte notre explication en français, mais en aucun cas ne la rendent inutile.

Pour coder la multiplication des polynômes en Caml, le plus difficile est de calculer la longueur du tableau résultat. On sait cependant que le monôme de plus haut degré du résultat a pour degré la somme des degrés des monômes de plus haut degré des polynômes multipliés. Or, le degré du monôme de plus haut degré d'un polynôme représenté par un tableau v de longueur l est $l - 1$. C'est donc, en Caml, `vect_length`$(v) - 1$. Par exemple, le polynôme $p = x^2 + 2x + 3$, est représenté par un tableau à trois cases et son monôme de plus haut degré est de degré 2. Mais le monôme de degré maximum du produit des polynômes p et q a pour degré la somme des degrés des monômes de degré maximum de p et q, soit (`vect_length`$(p) - 1) +$ (`vect_length`$(q) - 1$). On en déduit facilement que le tableau représentant $p \times q$ a pour longueur `vect_length`$(p) +$ `vect_length`$(q) - 1$.

Le programme Caml est bien plus court que ces explications :

```
# let multiplie_polynômes_pleins p q =
    let produit = make_vect (vect_length p + vect_length q - 1) 0 in
    for i = 0 to vect_length p - 1 do
      for j = 0 to vect_length q - 1 do
        produit.(i + j) <- p.(i) * q.(j) + produit.(i + j)
      done
    done;
    produit;;
multiplie_polynômes_pleins : int vect -> int vect -> int vect = <fun>
```

Notre programme effectue sans peine le produit des polynômes $(1 + 2x + 3x^2)$ et $(4 + 5x + 6x^2)$ que nous avions précédemment calculé à la main.

```
# imprime_polynôme_plein
    (multiplie_polynômes_pleins [|1; 2; 3|] [|4; 5; 6|]);;
4 + 13x + 28x^2 + 27x^3 + 18x^4- : unit = ()
```

Comme exemple plus consistant, calculons $(x+1)^2$ puis $(x+1)^4$ et $(x+1)^8$.

```
# let p = [| 1; 1|] in
  let p2 = multiplie_polynômes_pleins p p in
  let p4 = multiplie_polynômes_pleins p2 p2 in
  let p8 = multiplie_polynômes_pleins p4 p4 in
  print_string "(x + 1) ** 2 = ";
    imprime_polynôme_plein p2; print_newline ();
  print_string "(x + 1) ** 4 = ";
    imprime_polynôme_plein p4; print_newline ();
  print_string "(x + 1) ** 8 = ";
    imprime_polynôme_plein p8; print_newline ();;
(x + 1) ** 2 = 1 + 2x + x^2
(x + 1) ** 4 = 1 + 4x + 6x^2 + 4x^3 + x^4
(x + 1) ** 8 = 1 + 8x + 28x^2 + 56x^3 + 70x^4 + 56x^5 + 28x^6 + 8x^7 + x^8
- : unit = ()
```

3.5 Caractères et chaînes de caractères

En Caml les chaînes de caractères sont considérées comme des structures de données physiquement modifiables : les chaînes se comportent essentiellement comme des tableaux de caractères. On bénéficie ainsi d'un accès direct aux caractères et de la possibilité de modifier en place les caractères. La syntaxe de l'affectation et de l'accès aux caractères des chaînes est similaire à celle des tableaux, avec des crochets `[...]` à la place des parenthèses `(...)` :

Accès dans une chaîne s ::= s `.[` *indice* `]`
Modification d'un caractère d'une chaîne s ::= s `.[` *indice* `]` `<-` *expression*

Comme pour les tableaux, on parcourt donc complètement une chaîne de caractères par une boucle `for` commençant en 0 et finissant à la longueur de la chaîne moins 1, c'est-à-dire `string_length` $s - 1$. Par exemple, on calcule l'image miroir d'une chaîne en créant une chaîne de même longueur dont on écrit tous les caractères à l'aide d'une boucle `for` :

```
# let rev_string s =
   let l = string_length s in
   let res = make_string l ' ' in
   for i = 0 to l - 1 do
    res.[i] <- s.[l - 1 - i]
   done;
   res;;
rev_string : string -> string = <fun>
```

La primitive `make_string` permet de créer une chaîne de caractères d'une longueur donnée et initialisée avec un caractère donné : c'est donc l'analogue de `make_vect` pour les chaînes de caractères.

Au passage, cela donne une autre idée pour écrire une version simple et linéaire de la fonction `palindrome` de la section 2.1 :

```
# let palindrome s =
    let r = rev_string s in
    r = s;;
palindrome : string -> bool = <fun>
```

et même encore plus brièvement :

```
# let palindrome s = rev_string s = s;;
palindrome : string -> bool = <fun>
```

3.6 Les références

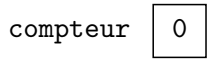
Les références sont des structures de données prédéfinies qui modélisent les cases mémoire de la machine. La propriété caractéristique des cases mémoire est qu'on peut les lire et les écrire : la lecture renvoie leur contenu courant, l'écriture change ce contenu. Les cases mémoire sont utilisées pour représenter des compteurs ou des accumulateurs, dont le contenu évolue au cours du calcul.

Lire et écrire les cases mémoire

Poursuivant l'analogie avec les cases mémoire, vous pouvez vous figurer une référence comme une *boîte* (la case mémoire) qui contient une valeur : vous pouvez placer quelque chose dans la boîte (écriture), ou demander à ouvrir la boîte pour examiner son contenu (lecture). Les références sont créées à l'aide de la construction `ref`(*val*), où *val* est la valeur initialement contenue dans la référence. Définissons par exemple un compteur qui vaut initialement 0 :

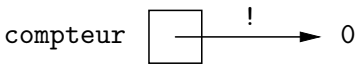
```
# let compteur = ref 0;;
compteur : int ref = ref 0
```

La valeur de `compteur` est donc une boîte contenant 0, qu'on peut représenter ainsi :



Le contenu courant d'une référence est renvoyé par l'opérateur de *déréférencement*, c'est-à-dire de lecture d'une référence, noté « `!` ». (Il ne faut pas confondre cette notation avec l'opérateur "factorielle" des mathématiques, que nous avons vu au chapitre 2, et qui se place après son argument ; le `!` Caml se place avant son argument.)

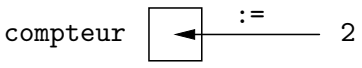
```
# !compteur;;
- : int = 0
```



On change le contenu d'une référence (écriture) en utilisant le symbole traditionnel de l'affectation « `:=` ».

```
# compteur := 2;;
- : unit = ()
```

L'affectation est, graphiquement, l'opération inverse de « `!` » :



Après l'affectation, le contenu de la boîte `compteur` est donc 2.

```
# !compteur;;
- : int = 2
```

Pour incrémenter `compteur`, nous devons ajouter 1 au contenu courant de la boîte :

```
# compteur := 1 + !compteur;;
- : unit = ()
# !compteur;;
- : int = 3
```

Une règle générale en Caml est que tous les objets du langage sont manipulables comme des valeurs sans restrictions particulières : on les passe en paramètre et on les rend en résultat, au même titre que les valeurs de base. Les références ne dérogent pas à cette règle. Nous pouvons donc passer des références en paramètre et définir une procédure qui incrémente le contenu de n'importe quelle référence contenant un entier (cette procédure est prédéfinie sous le nom `incr` dans le système Caml, mais nous en écrivons le code à titre d'exemple). La fonction prend une référence `c` en argument et modifie son contenu (`c := ...`) pour y mettre la valeur courante de la référence plus un (`1 + !c`) :

```
# let incrémente c = c := 1 + !c;;
incrémente : int ref -> unit = <fun>
# incrémente compteur; !compteur;;
- : int = 4
```

Les variables impératives

Un identificateur lié à une référence se comporte comme les *variables* des langages impératifs (C, Pascal, Ada), puisqu'on peut modifier à volonté le contenu de la référence. La seule différence est qu'en Caml il faut explicitement déréférencer l'identificateur à l'aide de l'opérateur `!`, pour en obtenir la valeur courante ; la distinction entre l'objet variable et la valeur courante de cet objet est donc plus nette.

Comme nous l'avions annoncé à la section 1.3, la définition d'un nom par la construction `let` est différente de l'affectation d'une variable. Nous sommes maintenant en mesure de comprendre cette différence, en comparant la *redéfinition* d'un identificateur par un nouveau `let` et l'affectation d'un identificateur lié à une référence :

```
# let x = 1;;
x : int = 1
# let f y = x + y;;
f : int -> int = <fun>
# let x = 2;;
x : int = 2
# f 0;;
- : int = 1
```

```
# let x = ref 1;;
x : int ref = ref 1
# let f y = !x + y;;
f : int -> int = <fun>
# x := 2;;
- : unit = ()
# f 0;;
- : int = 2
```

Dans la colonne de gauche, la redéfinition de `x` ne modifie en rien la valeur de `x` dans le corps de la fonction `f`. En revanche à droite, l'identificateur `x` est lié à une référence. La valeur de `!x` dans le corps de `f` change donc évidemment après l'affectation (cependant `x` est toujours lié à la même valeur : la même référence). On constate ainsi que les fonctions qui utilisent des références non locales sont susceptibles de changer dynamiquement de comportement, au gré des affectations des références qu'elles emploient.

3.7 Un programme utilisant des références

Un exemple réaliste d'utilisation des références nous est fourni par la fonction « factorielle », qui retourne le produit des nombres entiers inférieurs ou égaux à son argument. Nous en avions donné la définition récursive suivante au chapitre 2 :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{sinon.} \end{cases}$$

Voici une autre définition, dont nous admettrons qu'elle est mathématiquement équivalente :

$$\begin{aligned} 0! &= 1 \\ n! &= 1 \times 2 \times \cdots \times (n-1) \times n \end{aligned}$$

Cette définition avec trois petits points « . . . » est allusive et se traduit généralement par une implémentation sur machine à base de boucles et d'accumulateurs. Ainsi, on définira une référence pour accumuler les multiplications par les nombres plus petits que n, durant l'exécution d'une boucle allant de 1 à n. À chaque tour on multiplie le contenu actuel de l'accumulateur par l'indice de boucle courant (`accu := i * !accu`), si bien qu'à la fin de la boucle l'accumulateur contient le résultat voulu ; on renvoie donc son contenu (`!accu`).

```
# let fact n =
      if n = 0 then 1 else
       begin
         let accu = ref 1 in
         for i = 1 to n do accu := i * !accu done;
         !accu
       end;;
fact : int -> int = <fun>
```

```
# fact 10;;
- : int = 3628800
```

Une petite remarque : on peut supprimer le test `if n = 0` sans changer la sémantique de la fonction. En effet, lorsque `n` est nul, la boucle s'arrête instantanément, alors que la référence `accu` vaut 1, ce qui est le résultat correct dans ce cas. On obtient plus simplement :

```
# let fact n =
      let accu = ref 1 in
      for i = 1 to n do accu := i * !accu done;
      !accu;;
fact : int -> int = <fun>
```

Cet exemple nous permet de comparer à meilleur escient les styles impératif et fonctionnel. En effet, nous avons dû indiquer à la machine la suite des opérations à effectuer, en gérant explicitement les modifications successives du contenu de l'accumulateur `accu` : il s'agit vraiment ici d'un programme de style impératif. En revanche, souvenez-vous de notre précédente implémentation récursive de la fonction factorielle :

```
# let rec factorielle = function
    | 0 -> 1
    | n -> n * factorielle (n - 1);;
factorielle : int -> int = <fun>
```

Dans ce cas, on a presque l'impression d'avoir recopié la définition mathématique ! Vous comprenez aisément que le style impératif est plus descriptif du calcul à effectuer que le style fonctionnel. Le style impératif décrit *comment* faire le calcul, tandis que le style fonctionnel décrit *quoi* calculer. On dit que le style fonctionnel est davantage *déclaratif*. En effet, en écrivant la version fonctionnelle de `fact`, nous n'avons pas décrit comment faire : c'est le compilateur qui a géré lui-même l'enchaînement des calculs.

3.8 Récursivité et boucles

Cette section montre qu'une définition récursive peut cacher une boucle et qu'on peut définir une fonction récursive localement à une autre fonction. Rappelons la définition de la fonction `épelle` :

```
# let rec épelle_aux s i =
    if i < string_length s then
      begin
        print_char s.[i]; print_char ' ';
        épelle_aux s (i + 1)
      end;;
épelle_aux : string -> int -> unit = <fun>
# let épelle s = épelle_aux s 0;;
épelle : string -> unit = <fun>
```

Comme dans le cas des palindromes, si vous estimez que la fonction auxiliaire `épelle_aux` n'a pas d'intérêt en soi, puisqu'elle ne sert qu'à définir `épelle`, rien ne vous empêche de la définir localement à l'intérieur de la fonction `épelle` :

```
# let épelle s =
    let rec épelle_aux s i =
      if i < string_length s then
        begin
          print_char s.[i]; print_char ' ';
          épelle_aux s (i + 1)
        end in
    épelle_aux s 0;;
épelle : string -> unit = <fun>
# épelle "Bonjour";;
B o n j o u r - : unit = ()
```

Si l'on remarque alors que la chaîne `s` ne varie jamais pendant les appels à la fonction `épelle_aux`, on la supprime purement et simplement des arguments de `épelle_aux` (car `s` est liée à l'extérieur de `épelle_aux` par la fonction `épelle`). On obtient maintenant

```
# let épelle s =
    let rec épelle_aux i =
      if i < string_length s then
        begin
          print_char s.[i]; print_char ' ';
          épelle_aux (i + 1)
        end in
    épelle_aux 0;;
épelle : string -> unit = <fun>
```

On constate alors que la fonction `épelle_aux` n'est autre qu'une boucle `for` déguisée : son premier argument est 0, son dernier argument `string_length` $s - 1$, et `épelle_aux` augmente l'indice i de 1 à chaque appel récursif. On réécrit donc facilement `épelle` avec une boucle `for` :

```
# let épelle s =
    for i = 0 to string_length s - 1 do
      print_char s.[i]; print_char ' '
    done;;
épelle : string -> unit = <fun>
# épelle "Caml";;
C a m l - : unit = ()
```

C'est évidemment le cas aussi pour `compte` et `compte_à_rebours` :

```
# let compte n =
    for i = 1 to n do print_int i; print_string " " done;;
compte : int -> unit = <fun>
# let compte_à_rebours n =
    for i = n downto 1 do print_int i; print_string " " done;;
compte_à_rebours : int -> unit = <fun>
# compte 10; compte_à_rebours 10;;
1 2 3 4 5 6 7 8 9 10 10 9 8 7 6 5 4 3 2 1 - : unit = ()
```

On peut donc hésiter entre les deux styles de programmation. Chaque fois qu'une définition récursive code une boucle `for`, il faut sans conteste employer une boucle : les indices sont en tête de boucle, il n'y a pas besoin de fonction auxiliaire, c'est donc bien plus clair. En ce qui concerne l'efficacité, il n'est pas clair que l'une des versions

soit meilleure que l'autre : tout dépend du compilateur. Remarquez en particulier que les versions récursives ne font que des appels récursifs *terminaux*, c'est-à-dire placés en fin de fonction. Ce genre d'appels récursifs est très bien optimisé par les compilateurs Caml, qui les transforment automatiquement en boucles.

Le processus de réécriture d'une fonction récursive à l'aide de boucles, que nous avons vu ici sur des exemples, s'appelle la « dérécursivation ». Cette opération est souvent difficile ; essayez par exemple de dérécursiver la fonction `hanoi`. *A contrario*, les fonctions récursives permettent d'écrire facilement les boucles les plus complexes, en particulier lorsque la boucle comporte plusieurs sorties possibles ou lorsque l'indice ne varie pas de façon uniforme, ou simplement lorsqu'il faut retourner une valeur significative (différente de `()`) à la fin de la boucle. Cependant la récursivité ne se limite certainement pas à coder des boucles : c'est un outil extrêmement puissant, comme vous l'avez constaté avec le jeu de Hanoi. Nous en verrons encore beaucoup d'exemples non triviaux dans la suite.

3.9 Règle d'extensionnalité

Il nous faut faire ici une remarque importante, que nous utiliserons quelques fois. Il s'agit d'un exemple de règle de *calcul sur les programmes* qui permet à l'occasion de simplifier les programmes. Ce genre de règles constitue l'équivalent informatique des règles de simplification des mathématiques. Cette règle est très facile : elle stipule qu'il est inutile de définir une fonction qui se contente d'en appeler immédiatement une autre.

```
# let successeur x = x + 1;;
successeur : int -> int = <fun>
# let succ = function x -> successeur x;;
succ : int -> int = <fun>
```

La fonction `succ` applique immédiatement la fonction `successeur` à son argument `x` ; elle retournera donc forcément toujours les mêmes valeurs que la fonction `successeur` : c'est la même fonction que `successeur`. On dit que la fonction `succ` est *égale* à la fonction `successeur`. D'ailleurs, on aurait très bien pu la définir par :

```
# let succ = successeur;;
succ : int -> int = <fun>
```

Plus généralement on a,

Pour toute fonction f, `function` x `->` $f\ x$ est équivalent à f

De la même façon qu'en mathématiques les règles du calcul algébrique permettent de remplacer $x - x$ par 0, nous utiliserons la règle précédente pour écrire `successeur` au lieu du programme `function x -> successeur x`. Nous aurons ainsi simplifié notre programme en utilisant une règle universelle, sans même avoir besoin de réfléchir au contenu du programme.

On se convainc facilement de la validité de la règle précédente en remarquant que les deux fonctions retournent toujours le même résultat quel que soit leur argument. En effet, (`function` x `->` $f\ x$) (y) s'évalue comme $f\ y$ (en remplaçant x par y dans

$f\ x$). On peut également considérer que cette règle est la simple traduction de la définition mathématique des fonctions : la fonction f est celle qui à tout élément x de son ensemble de départ fait correspondre son image par f, c'est-à-dire $f\ (x)$. La fonction f est donc la correspondance $x \mapsto f(x)$, c'est exactement ce que traduit la règle « $f =$ `function` x `->` $f\ x$ ».

Cette règle est appelée « règle d'extensionnalité » ou encore « règle η » (la lettre grecque η se prononce « êta »). Quand on remplace f par l'expression plus complexe `function` x `->` $f\ x$, on dit qu'on fait une η-expansion. Inversement, quand on simplifie `function` x `->` $f\ x$ en f, on fait une η-contraction. La plupart du temps, nous emploierons la règle η dans le sens de la simplification (η-contraction).

Les conventions syntaxiques de définition et d'application des fonctions se combinent avec la règle η pour établir que :

`let` $g\ x$ `=` $f\ x$ `;;` est équivalent à `let` g `=` f `;;`

En effet, la convention syntaxique des définitions de fonctions nous permet d'écrire :

`let` g `=` `function` x `->` $f\ x$ au lieu de `let` $g\ x$ `=` $f\ x$ `;;`

Puis, en appliquant la règle η, on simplifie `function` x `->` $f\ x$ en f et l'on obtient :

`let` g `=` f `;;`

Vous pouvez constater que nous avons ici raisonné sur un petit programme indépendamment de sa signification : on ne sait pas ce que fait f ni pourquoi on définit g. C'est une force de connaître ce genre de raisonnements très généraux qui s'appliquent dans toutes les situations.

Nous n'irons pas plus loin dans ces considérations un peu complexes et formelles. La règle η restera notre seule règle de calcul sur les programmes. Son nom grec ne doit pas effrayer, car la règle η est intuitivement très naturelle : elle stipule simplement que la fonction qui à x associe $f(x)$ est la fonction f, et réciproquement.

3.10 Effets et évaluation

Cette section peut être sautée en première lecture.

Effets et appels de fonctions

Avec les effets, nous sortons du monde intemporel des mathématiques pour entrer dans un monde qui connaît les notions de passé et d'avenir et dans lequel l'enchevêtrement des calculs a son importance. Pour calculer une expression sans effets comme `1 * 2 + 3 * 4`, peu importe l'ordre dans lequel on effectue les calculs : qu'on commence par calculer `1 * 2` ou `3 * 4`, le résultat est identique. Il n'en va pas de même si l'on mélange calculs et effets. Pour mettre ce phénomène en évidence, ajoutons des effets dans le calcul de `1 * 2 + 3 * 4` en y mêlant des impressions au terminal. Remplaçons d'abord les nombres par des séquences qui les impriment, par exemple remplaçons `1` par `(print_int 1; 1)`. (On doit obligatoirement parenthéser une séquence pour l'inclure dans une opération.) Puis effectuons le calcul en commençant par l'une ou l'autre de ses sous-expressions : d'abord `1 * 2`, puis `3 * 4`.

```
# let un_fois_deux = (print_int 1; 1) * (print_int 2; 2) in
  let trois_fois_quatre = (print_int 3; 3) * (print_int 4; 4) in
  un_fois_deux + trois_fois_quatre;;
2143- : int = 14
# let trois_fois_quatre = (print_int 3; 3) * (print_int 4; 4) in
  let un_fois_deux = (print_int 1; 1) * (print_int 2; 2) in
  un_fois_deux + trois_fois_quatre;;
4321- : int = 14
```

On n'obtient évidemment pas les mêmes impressions à l'écran. Laissons le compilateur nous dévoiler l'ordre qu'il choisit :

```
# (print_int 1; 1) * (print_int 2; 2) +
  (print_int 3; 3) * (print_int 4; 4);;
4321- : int = 14
```

Un autre compilateur aurait pu choisir un autre ordre. La conclusion de cette expérience est qu'il ne faut jamais mélanger effets et appels de fonctions, car on ne sait pas alors déterminer le moment où les effets vont intervenir. En effet, l'ordre d'évaluation des arguments d'une fonction n'est pas garanti par le langage. Seules la séquence, l'alternative et la construction `let` ont un ordre d'évaluation déterminé. Ce n'est pas étonnant pour la séquence, puisque c'est son rôle de fixer l'ordre d'évaluation de deux expressions. Pour l'alternative, il est clair qu'on ne peut décider la branche à choisir qu'après avoir évalué la condition. En ce qui concerne le `let`, on évalue toujours l'expression définissante d'abord : dans `let` $x = e_1$ `in` e_2 on évalue e_1 avant e_2, garantissant ainsi que la valeur de l'identificateur x est connue pendant le calcul de l'expression e_2.

Effets et règle η

La règle η du paragraphe précédent stipule que l'expression (`function` x `->` $f\ x$) est équivalente à f. Nous avons précisé que cette règle s'applique quand f est une fonction. En fait, cette règle s'étend facilement au cas où f est une expression quelconque, mais seulement dans le cas où cette expression f ne produit pas d'effets. Insistons : la règle est parfaitement correcte pour toute fonction f, que f produise des effets ou non. De plus cette règle s'étend à toute expression, pourvu que cette expression ne produise aucun effet. Il est pourtant impossible d'étendre la règle à une expression quelconque, car certaines expressions produisant des effets l'invalident. Il est si tentant d'utiliser cette règle étendue sans vérifier que l'expression impartie est sans effets que nous pensons nécessaire d'étudier un exemple où les effets empêchent de l'utiliser sous peine d'erreur. Définissons une fonction `f` qui incrémente son argument, puis retourne une fonction en résultat :

```
# let f x = incr x; (function z -> z + 1);;
f : int ref -> int -> int = <fun>
```

Puis nous définissons une autre fonction, `g`, qui appelle `f` avec la variable déjà définie `compteur`. Nous prenons également bien soin de suivre l'évolution du contenu de la référence `compteur`.

```
# !compteur;;
- : int = 4
```

```
# let g y = f compteur y;;
g : int -> int = <fun>
# !compteur;;
- : int = 4
```

On constate que la définition de `g` n'a pas modifié la valeur de `compteur`, ce qui semble normal. En revanche, chaque fois qu'on appelle la fonction `g` avec une valeur `v`, on évalue l'expression `f compteur v`, si bien que la valeur de `compteur` est incrémentée à chaque appel de `g`, ce qui semble toujours un comportement raisonnable.

```
# g 0;;
- : int = 1
# !compteur;;
- : int = 5
```

Mais supposez maintenant que nous utilisions la règle η pour simplifier la définition de `g` en supprimant l'argument `y`. C'est bien sûr interdit, car nous utiliserions la règle avec l'expression « `f compteur` » qui n'est pas une fonction (c'est une application) et qui de surcroît produit des effets. Pour montrer que le comportement du programme changerait, faisons-le tout de même, en suivant toujours soigneusement l'évolution de la valeur de `compteur`.

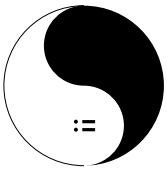
```
# let g = f compteur;;
g : int -> int = <fun>
# !compteur;;
- : int = 6
```

On constate que la valeur de `compteur` a été modifiée *en définissant* `g`. En effet, pour définir `g` on applique maintenant `f` à `compteur` et cela produit tout naturellement un effet sur `compteur` *pendant la définition* de `g`, puisque l'évaluation de `f compteur` provoque l'exécution de la séquence `incr x; (function z -> z + 1)` où `x` est lié à `compteur`. On incrémente donc `compteur` et l'on renvoie la fonction. Maintenant, appelons `g` comme précédemment :

```
# !compteur;;
- : int = 6
# g 0;;
- : int = 1
# !compteur;;
- : int = 6
```

La valeur de `compteur` n'est plus modifiée à l'appel de `g`. En effet, `g` a maintenant pour valeur la fonction `function z -> z + 1` qui ne fait bien sûr aucun effet sur la valeur de `compteur`. Par la suite, les appels à `g` n'incrémenteront donc plus jamais `compteur`.

Ce comportement est dû à l'effet retard induit par les paramètres de fonction : quand on définit `g` avec un argument explicite `y`, il faut attendre l'application de `g` à une valeur pour commencer l'exécution du corps de `g`, donc l'évaluation de `f compteur y`. Dans le cas d'une fonction définie sans argument (par un calcul), les effets interviennent immédiatement puisqu'il n'y a aucune valeur de paramètre à attendre.



4
Fonctionnelles et polymorphisme

Où l'on apprend qu'il y a des fonctions qui fabriquent des fonctions et des types qui remplacent n'importe quoi.

Quoique réputées difficiles, les notions de polymorphisme et de pleine fonctionnalité s'introduisent très naturellement en Caml et vous constaterez dans ce chapitre qu'elles ne sont pas si ardues. Si vous n'avez jamais entendu parler de polymorphisme, lisez la première section. Les sections suivantes, jusqu'à la section 4.7, montrent le polymorphisme à l'œuvre dans des situations pratiques. Au passage (section 4.3), nous ferons le point sur l'algèbre de types de Caml.

4.1 Notion de polymorphisme

Étymologiquement, *polymorphe* signifie plusieurs (*poly*) formes (*morphe*). On emploie ce mot par exemple en psychologie pour parler de pervers polymorphes (pervers qui ne sont pas fixés sur une forme précise de perversion, ce qui est un stade normal de développement psychologique de l'enfant), ou bien en médecine pour des maladies qui entraînent des symptômes variables, ou des virus dont l'aspect varie. En informatique, ce terme désigne des objets ou des programmes qui peuvent servir *sans modifications* dans des contextes très divers. Par exemple, une fonction de tri d'objets sera *monomorphe* si elle ne s'applique qu'à un seul type d'objets (par exemple les entiers) et *polymorphe* si elle s'applique à tous les types d'objets qu'on peut comparer pour les ranger du plus petit au plus grand. Dans ce dernier cas, le même programme de tri s'appliquera sans modifications à des entiers (comparaison $\leq$), à des nombres flottants (comparaison $\leq$ des flottants) et à des chaînes de caractères (ordre du dictionnaire). Du point de vue du typage, cela signifie que la fonction de tri pourra être employée avec plusieurs types différents.

Le polymorphisme n'est pas l'apanage des fonctions : certaines valeurs non fonctionnelles peuvent aussi être utilisées avec plusieurs « formes », c'est-à-dire plusieurs types. Les exemples se trouvent du côté des structures de données comme les tableaux et les listes : clairement, un tableau de nombres entiers ne pourra pas être employé avec un autre type, mais vous admettrez facilement que le tableau vide (le tableau à zéro élément) peut être vu comme un tableau d'entiers aussi bien que comme un tableau

de chaînes. Nous allons nous intéresser d'abord aux fonctions polymorphes, parce que c'est dans le domaine des fonctions que cette notion est la plus naturelle et la plus facile à appréhender.

Pour exprimer le polymorphisme dans les expressions de types, nous avons besoin d'une notion de types qui puissent remplacer plusieurs types différents : ce sont les *paramètres de type*, qu'on distingue syntaxiquement des types ordinaires en les faisant précéder d'une apostrophe (`'`). Par exemple, `'a` est un paramètre de type nommé `a`.

Le polymorphisme de Caml est techniquement qualifié de *paramétrique*. Intuitivement cela signifie que ce polymorphisme fonctionne en « tout ou rien ». La signification d'un paramètre de type est de remplacer *n'importe quel autre type* et non pas *un certain nombre* d'autres types. On n'aura donc pas de programmes Caml uniquement valables pour un ensemble déterminé de types. Par exemple, il n'y a aucun moyen de définir une fonction qui s'appliquerait uniquement à des entiers et des chaînes de caractères (et qui aurait donc un type du genre `(int` *ou* `string) -> ...`) Un programme Caml s'applique soit à *tous les types possibles*, soit à *un seul et unique type*. Dans le premier cas le type du programme comporte un paramètre (par exemple `'a -> ...`), dans le second cas il n'en comporte pas (par exemple `int -> ...`). Voyons un premier exemple :

```
# let successeur x = x + 1;;
successeur : int -> int = <fun>
```

La fonction est monomorphe, comme on s'y attend : elle ne s'applique qu'à des entiers, puisqu'on doit faire une addition avec son argument. Mais supposons qu'on supprime l'addition qui entraîne cette contrainte sur l'argument `x` et qu'on renvoie directement 1.

```
# let fonction_un x = 1;;
fonction_un : 'a -> int = <fun>
```

La fonction `fonction_un` est maintenant polymorphe : elle ne fait rien de son argument, on peut donc l'appliquer à n'importe quoi.

```
# fonction_un 2;;
- : int = 1
```

```
# fonction_un "oui";;
- : int = 1
```

```
# fonction_un true;;
- : int = 1
```

Contrairement à ce que suggère l'exemple `fonction_un`, une fonction polymorphe peut utiliser son argument, par exemple en le renvoyant tel quel. Nous supprimons encore une fois l'addition dans le code de `successeur`, mais cette fois nous renvoyons `x` au lieu de `1` :

```
# let identité x = x;;
identité : 'a -> 'a = <fun>
```

Nous obtenons encore une fonction polymorphe. Notez que le type de la fonction `identité` indique à juste titre que le type du résultat est exactement celui de l'argument. Le paramètre `'a` remplace n'importe quel type, en particulier `string` ou `int`, légitimant ainsi l'emploi de `identité` avec le type `string -> string`, et aussi avec le type `int -> int` :

```
# identité "non";;
- : string = "non"
```

```
# identité 1;;
- : int = 1
```

Ce mécanisme de remplacement d'un paramètre de type par un type quelconque s'appelle la *spécialisation*. Nos deux exemples consistent donc à spécialiser `'a` en `string`, puis en `int`. On n'est pas obligé de spécialiser un paramètre avec un type de base, comme nous l'avons fait jusqu'à présent ; on le spécialise tout aussi bien avec un type complexe, par exemple `int -> int`. Dans le cas de la fonction `identité`, on obtient le type `(int -> int) -> (int -> int)`. Cela suggère d'appeler la fonction `identité` sur un argument qui est lui-même une fonction ; et pourquoi pas la fonction `successeur` ?

```
# identité successeur;;
- : int -> int = <fun>
```

La fonction `identité` renvoie toujours son argument sans modification ; elle renvoie donc tout simplement la fonction `successeur` quand on l'applique à `successeur`. Par exemple :

```
# let success = identité successeur;;
success : int -> int = <fun>
# success 3;;
- : int = 4
```

4.2 Fonctions d'ordre supérieur

Les fonctions d'ordre supérieur sont des fonctions dont les arguments ou les résultats sont eux-mêmes des fonctions. Une fonction d'ordre supérieur est encore appelée une *fonctionnelle*. Ces fonctions sont souvent polymorphes et surtout employées avec des structures de données plus complexes que les types de base. Il n'est pas nécessaire de tout connaître sur le polymorphisme et les fonctions d'ordre supérieur pour les utiliser. Pour le lecteur plus intéressé par les programmes que par la théorie, il suffit donc de lire cette section pour comprendre le reste des exemples de ce livre.

Nous commencerons par les fonctions qui renvoient d'autres fonctions, puis nous verrons les fonctions dont les arguments sont fonctionnels. Les exemples présentés ici seront sans doute un peu artificiels, puisque nous ne disposons pas encore des outils suffisants pour montrer des utilisations réalistes de fonctionnelles sur des structures de données complexes.

Fonctions retournant des fonctions

Supposez que nous voulions définir la fonction `fois_x` qui, étant donné un entier `x`, fabrique la fonction qui multipliera par `x`. La définition en pseudo-code Caml serait :

`let fois_x (x) =` « la fonction qui multiplie par `x` »`;;`

Nous écrivons donc du vrai code pour la périphrase « la fonction qui multiplie par `x` » :

```
# let fois_x (x) = (function y -> x * y);;
fois_x : int -> int -> int = <fun>
```

La fonction `fois_x` est une fonctionnelle : quand on l'applique à un argument elle fabrique une fonction qu'elle renvoie en résultat ! Ainsi, on obtient la multiplication par 2 en appliquant `fois_x` à 2.

```
# let double = fois_x (2);;
double : int -> int = <fun>
# double (3);;
- : int = 6
```

On constate ainsi qu'on a défini la fonction `double` non pas en donnant les arguments et le corps de la fonction (par la construction `function`), mais par un calcul : nous avons *calculé une fonction*. Il n'est d'ailleurs même pas nécessaire de donner un nom à la fonction `double` pour calculer `double (3)` : nous pouvons directement appliquer la fonction calculée `fois_x (2)` à la valeur `3`.

```
# (fois_x (2)) (3);;
- : int = 6
```

La syntaxe de Caml autorise une écriture encore simplifiée de cette expression, sans aucune parenthèse. Rappelons que les conventions syntaxiques pour l'application des fonctions en Caml stipulent en effet que

- `f x` est lu comme `f (x)`
- `f x y` est lu comme `(f x) y`.

En employant la première règle on obtient d'abord `(fois_x 2) 3`, puis en appliquant la seconde, on obtient la forme la plus simple `fois_x 2 3`.

```
# fois_x 2 3;;
- : int = 6
```

Fonctions dont les arguments sont des fonctions

Faisons un pas de plus : définissons une fonction avec un argument fonctionnel. Par exemple, la fonctionnelle `double_le_résultat_de` qui double le résultat de sa fonction argument : étant donnés une fonction `f` et un argument `x` pour `f`, `double_le_résultat_de` retourne la valeur `2 * f(x)`, c'est-à-dire `double (f x)`.

```
# let double_le_résultat_de (f : int -> int) =
      function x -> double (f x);;
double_le_résultat_de : (int -> int) -> int -> int = <fun>
```

Avec cette fonctionnelle nous pouvons à nouveau *calculer des fonctions*. Par exemple, la fonction qui retourne le double du successeur de son argument, `function x -> 2 * (x + 1)`, se calcule en appliquant la fonctionnelle `double_le_résultat_de` à la fonction `successeur`. C'est un calcul tout à fait normal, si ce n'est qu'il implique des fonctions et que le résultat est en fait une fonction.

```
# let double_du_successeur = double_le_résultat_de successeur;;
double_du_successeur : int -> int = <fun>
# double_du_successeur 3;;
- : int = 8
```

La fonctionnelle « sigma »

À l'occasion de la définition des opérations sur les polynômes, nous avons vu la notation mathématique Σ, définie par :

$$\sum_{i=0}^{n} \mathit{formule}\,(i) \;=\; \mathit{formule}\,(0) + \mathit{formule}\,(1) + \cdots + \mathit{formule}\,(n).$$

Avec le vocabulaire des langages fonctionnels, la notation Σ n'est rien d'autre qu'une fonctionnelle qui prend en argument une fonction représentant *formule* et l'applique successivement à $0, 1, \ldots, n$. Il est très facile de définir cette fonctionnelle en Caml, à l'aide d'une boucle et d'un accumulateur :

```
# let sigma formule n =
      let résultat = ref 0 in
      for i = 0 to n do résultat := !résultat + formule (i) done;
      !résultat;;
sigma : (int -> int) -> int -> int = <fun>
```

ou même à l'aide d'une fonction récursive

```
# let rec sigma formule n =
      if n <= 0 then 0 else formule n + sigma formule (n - 1);;
sigma : (int -> int) -> int -> int = <fun>
```

Nous pouvons maintenant faire calculer par Caml les exemples que nous avions donnés. Si la formule est réduite à i, on obtient la somme des nombres de 0 à n :

$$\sum_{i=0}^{n} i \;=\; 0 + 1 + \cdots + n.$$

Cette formule correspond à appeler la fonctionnelle `sigma` avec l'argument `(function i -> i)`.

```
# sigma (function i -> i) 10;;
- : int = 55
```

```
# sigma identité 10;;
- : int = 55
```

De même la somme des carrés des nombres entre 0 et n,

$$\sum_{i=0}^{n} i^2 \;=\; 0^2 + 1^2 + \cdots + n^2,$$

s'obtient par application de `sigma` à l'argument `(function i -> i * i)`.

```
# sigma (function i -> i * i) 10;;
- : int = 385
```

4.3 Typage et polymorphisme

Synthèse du type le plus général

Comme nous l'avons déjà dit, le compilateur de Caml donne un type à chaque phrase entrée par l'utilisateur ; cette *inférence de types* ne nécessite aucune participation de

l'utilisateur : elle se produit automatiquement sans nécessité d'indiquer les types dans les programmes. Connaissant les types des valeurs de base et des opérations primitives, le contrôleur de types produit un type pour une phrase en suivant des *règles de typage* pour les constructions du langage comme la définition et l'application des fonctions. De plus, le type inféré contient le plus petit ensemble de contraintes nécessaires au bon déroulement de l'exécution du programme (ici, « bon déroulement » signifie qu'il n'y aura pas d'erreurs de type à l'exécution). On dit que le contrôleur de type trouve le type le plus général de chaque expression (notion introduite par Robin Milner en 1978). Par exemple, la fonction `successeur` reçoit le type `int -> int` parce que son argument *doit* être un entier, puisqu'on lui ajoute 1. En revanche la fonction `identité` a le type `'a -> 'a` parce qu'il n'y a pas de contrainte sur son argument. Le polymorphisme s'introduit donc naturellement à partir de l'absence de contraintes sur le type d'un argument ou d'une valeur. Par exemple, rappelons la définition de la fonctionnelle `double_le_résultat_de` :

```
# let double_le_résultat_de (f : int -> int) =
      function x -> double (f x);;
double_le_résultat_de : (int -> int) -> int -> int = <fun>
```

L'argument `f` devait être une fonction des entiers vers les entiers, à cause de la contrainte de type `(f : int -> int)`, explicitement écrite dans le programme. Mais si nous retirons cette contrainte de type, nous obtenons une fonctionnelle plus générale :

```
# let double_le_résultat_de f = function x -> double (f x);;
double_le_résultat_de : ('a -> int) -> 'a -> int = <fun>
```

La fonctionnelle devient polymorphe, car le contrôleur de type a découvert que `f` devait seulement renvoyer un entier en résultat, mais qu'il n'est nullement obligatoire qu'elle prenne un entier en argument. Voici un exemple où `f` reçoit une chaîne de caractères :

```
# let double_de_la_longueur = double_le_résultat_de string_length;;
double_de_la_longueur : string -> int = <fun>
# double_de_la_longueur "Caml";;
- : int = 8
```

Le polymorphisme découle donc de l'absence de contraintes sur une valeur. Cela explique pourquoi un paramètre de type peut être remplacé sans risque d'erreurs par n'importe quel type, y compris un type lui-même polymorphe. Par exemple, on applique la fonction `identité` à elle-même en l'employant avec le type `('a -> 'a) -> ('a -> 'a)` :

```
# let id x = (identité identité) x;;
id : 'a -> 'a = <fun>
```

Puisque la fonction `identité` renvoie toujours son argument, `(identité identité)` s'évalue en `identité`, et la fonction `id` est donc tout simplement égale à la fonction `identité`.

L'algèbre des types de Caml

Nous allons maintenant préciser davantage l'ensemble des types qu'utilise le système Caml, ce qu'on nomme techniquement son algèbre des types. Tout type Caml entre dans l'une des catégories suivantes :

- Types de base (comme `int` ou `string`).
- Types composites (comme `int -> int` ou `int vect`).
- Paramètres de type (comme `'a`).

Les types composites sont construits avec des *constructeurs de types*, tels que la flèche `->`. Étant donnés deux types t_1 et t_2, le constructeur de type flèche construit le type t_1 `->` t_2, qui est le type des fonctions ayant un argument du type t_1 et rendant un résultat du type t_2, autrement dit les fonctions de t_1 dans t_2. Remarquons que le constructeur flèche est un opérateur binaire (deux arguments) et infixe (situé entre ses arguments, comme l'est le symbole de l'addition `+`). En revanche, le constructeur de types `vect` est unaire, puisqu'à partir d'un unique type t_1, il construit le type t_1 `vect`. Ce constructeur est postfixe, c'est-à-dire placé après son argument. Tous les constructeurs de types unaires sont postfixés en Caml. Par extension, les types n'ayant pas d'arguments (`int` par exemple) sont appelés constructeurs de types constants.

Les paires

Il existe un autre constructeur de type binaire et infixe dont nous n'avons pas encore parlé : le constructeur prédéfini « `*` ». Étant donnés deux types t_1 et t_2, la notation t_1 `*` t_2 est donc un type. C'est le produit cartésien des types t_1 et t_2. Il dénote le type des couples d'un élément du type t_1 avec un élément du type t_2. En mathématiques, le produit cartésien de deux ensembles A et B est l'ensemble des couples (x, y) tels que x est élément de A et y élément de B. Le produit cartésien de A et B est noté $A \times B$. Cette analogie avec la notation de la multiplication est aussi employée en Caml, d'où le symbole `*` dans les types.

Les valeurs de types produit se notent comme en mathématiques : on écrit les deux éléments du couple entre parenthèses et séparés par une virgule. Une petite différence d'appellation cependant : en informatique on parle plus volontiers de paires que de couples. De plus, en Caml, les parenthèses autour des paires ne sont pas toujours strictement nécessaires.

```
# (1, 2);;
- : int * int = 1, 2
```

Les paires sont aussi utilisées en tant qu'arguments ou résultats de fonctions.

```
# let addition (x, y) = x + y;;
addition : int * int -> int = <fun>
# addition (1, 2);;
- : int = 3
```

À l'aide de paires, on écrit des fonctions qui rendent plusieurs résultats. Par exemple, la fonction suivante calcule simultanément le quotient et le reste d'une division entière :

```
# let quotient_reste (x, y) = ((x / y), (x mod y));;
quotient_reste : int * int -> int * int = <fun>
# quotient_reste (5, 3);;
- : int * int = 1, 2
```

Les notations pour les paires se généralisent aux triplets, aux quadruplets, et en fait aux n-uplets pour n'importe quel nombre d'éléments n. Par exemple, `(1, 2, 3)` est un triplet d'entiers et possède le type `int * int * int`.

4.4 Curryfication

À proprement parler, une fonction prenant une paire comme argument ne possède quand même qu'un seul argument et non pas deux. La fonction `addition` ci-dessus, qui prend un seul argument qui se trouve être une paire, est différente de la fonction `add` suivante, qui prend deux arguments.

```
# let add x y = x + y;;
add : int -> int -> int = <fun>
```

Du point de vue pratique, la différence est minime, il est vrai. D'un point de vue technique, une fonction qui reçoit ses arguments un par un (comme `add`) est dite *curryfiée*. En revanche, une fonction qui reçoit tous ses arguments à la fois sous la forme d'une paire ou plus généralement d'un n-uplet de valeurs est dite *non curryfiée*. Le néologisme « curryfier » n'est pas une allusion à la cuisine indienne, mais un hommage au logicien Haskell Curry.

Application partielle

La différence essentielle entre `add` et `addition` tient dans la manière de les appliquer : il est légal d'appliquer la fonction `add` à un seul argument, obtenant ainsi une fonction comme résultat, tandis que la fonction `addition` doit forcément recevoir ses deux entiers en même temps. Cette capacité des fonctions curryfiées de ne recevoir qu'un certain nombre de leurs arguments permet l'*application partielle*. Par exemple, en appliquant (partiellement) `add` à l'entier `1`, on obtient la fonction `successeur`.

```
# let successeur = add 1;;
successeur : int -> int = <fun>
# successeur 3;;
- : int = 4
```

Curryfication et type flèche

Une fonction curryfiée est donc un cas particulier de fonctionnelle, puisqu'elle permet de créer d'autres fonctions, en fixant certains de ses arguments. Cette propriété est en fait inscrite dans le type d'une fonction curryfiée. Par exemple, le type de `add` est `int -> int -> int`. Or, le constructeur de type `->` associe à droite, ce qui signifie que le type de `add` n'est autre que `int -> (int -> int)`. Cette écriture explicitement parenthésée indique clairement que `add` est une fonctionnelle : étant donné un entier, `add` retourne une autre fonction dont le type est justement `(int -> int)`. Cela paraît difficile à comprendre au premier abord, mais c'est simplement une autre manière de voir des phrases aussi simple que « ajouter 2 au résultat précédent », qui signifie en fait : utiliser l'addition avec l'un des arguments fixé à 2 et appliquer cette fonction au résultat précédent. En Caml, cela correspondrait à évaluer :

```
(add 2) (« résultat précédent »);;
```

Une autre approche féconde est de considérer `add` comme une fonction générique, qui permet d'obtenir la famille de toutes les fonctions qui ajoutent une constante à leur argument (et qui sont donc de type `int -> int`). Par exemple, la fonction `add_3`, qui ajoute `3` à son argument, est définie par :

```
# let add_3 = add 3;;
add_3 : int -> int = <fun>
```

L'application partielle d'une fonction curryfiée pour fixer certains de ces arguments se justifie lorsque la fonction est très générale. Dans ce cas, cette opération de spécialisation permet de retrouver des fonctions intéressantes en elles-mêmes. Nous en verrons un exemple avec le tri, où fixer l'argument fonctionnel correspondant à la comparaison permet de définir le tri en ordre croissant ou le tri en ordre décroissant.

De cette étude des fonctions curryfiées, retenons que le constructeur de type $\rightarrow$ est *associatif à droite*, ce qui signifie tout simplement que :

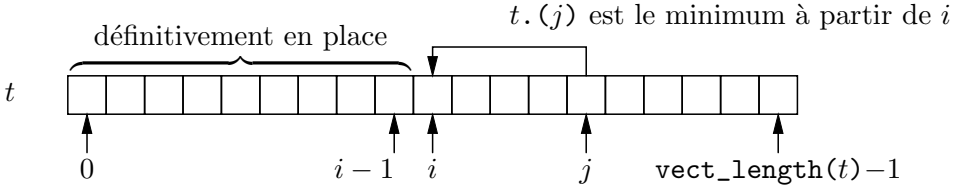
$$t_1 \rightarrow t_2 \rightarrow t_3 \quad \text{est équivalent à} \quad t_1 \rightarrow (t_2 \rightarrow t_3)$$

4.5 Une fonctionnelle de tri polymorphe

Le polymorphisme n'est pas réservé au style fonctionnel. Pour le montrer, nous définissons ici une procédure qui trie en place un tableau. La procédure ne calcule pas de valeurs, mais modifie l'ordre des éléments dans le tableau. Nous généralisons ensuite cette procédure pour la rendre applicable à tout type d'ordre.

Trier des tableaux en ordre croissant

L'algorithme de tri employé ici repose sur une idée très simple : on cherche le minimum des éléments du tableau et on le met à la première place. Puis on cherche le minimum du reste du tableau et on le met à la seconde place. Il suffit d'itérer ce procédé jusqu'à avoir parcouru complètement le tableau. Ce tri s'appelle le *tri par sélection*. Son principe s'illustre graphiquement par le schéma suivant, qui montre la $i^{\text{ième}}$ étape du tri par sélection, consistant à déplacer $t.(j)$ en $t.(i)$, où j est l'indice tel que $t.(j)$ est le plus petit des éléments $t.(i)$, $t.(i+1)$, ...



Pour traduire cet algorithme en Caml, nous avons besoin d'une procédure auxiliaire qui se charge d'échanger deux éléments dans un tableau. Il est bien connu que cette tâche présente une petite difficulté : il faut garder une copie d'un des éléments, puisqu'on perd cet élément quand on écrit l'autre à sa place. Cela mérite bien une procédure indépendante de la fonction de tri. La procédure d'échange est naturellement polymorphe : elle peut s'appliquer à tout type de tableau puisqu'elle effectue une opération indépendante de la nature des éléments du tableau.

```
# let échange t i j = (* échange les éléments i et j de t *)
    let élément_i = t.(i) in
    t.(i) <- t.(j);
```

```
    t.(j) <- élément_i;;
échange : 'a vect -> int -> int -> unit = <fun>
```

Il n'y a plus de difficulté à écrire en Caml le tri par sélection :

```
# let tri t =
    for i = 0 to vect_length t - 2 do
      let min = ref i in
      for j = i + 1 to vect_length t - 1 do
        if t.(j) <= t.(!min) then min := j
      done;
      échange t i !min
    done;;
tri : 'a vect -> unit = <fun>
```

Généralisation à tout type d'ordre

Pour rendre cette procédure polymorphe encore plus générale, il suffit de remarquer que c'est l'emploi de la primitive `<=` pour comparer les éléments du tableau qui impose le rangement par ordre croissant. Il suffit donc de passer la fonction de comparaison en argument pour trier selon toute sorte d'ordres. Lorsqu'on passe ainsi en argument une fonction utilisée dans le corps d'une définition, on dit qu'on « abstrait » la fonction. Dans le cas du tri on abstrait donc la comparaison et la fonction `tri` prend alors un argument supplémentaire, `ordre`, une fonction à deux arguments x et y renvoyant `true` si x est inférieur ou égal à y et `false` sinon (ces fonctions sont appelées *prédicats*).

```
# let tri ordre t =
    for i = 0 to vect_length t - 2 do
      let min = ref i in
      for j = i + 1 to vect_length t - 1 do
        if ordre t.(j) t.(!min) then min := j
      done;
      échange t i !min
    done;;
tri : ('a -> 'a -> bool) -> 'a vect -> unit = <fun>
```

Trions par ordre croissant ou décroissant un tableau d'entiers, en changeant simplement l'ordre :

```
# let t = [|3; 1; 2|] in
  tri (function x -> function y -> x <= y) t; t;;
- : int vect = [|1; 2; 3|]
# let t = [|3; 1; 2|] in
  tri (function x -> function y -> x >= y) t; t;;
- : int vect = [|3; 2; 1|]
```

Ce n'est pas plus difficile pour les tableaux de chaînes de caractères, en utilisant les fonctions prédéfinies de comparaison `ge_string` (supérieur ou égal sur les chaînes) ou `le_string` (inférieur ou égal), qui comparent deux chaînes dans l'ordre lexicographique, c'est-à-dire l'ordre du dictionnaire.

```
# let t = [|"Salut "; "les "; "copains!"|] in
  tri (function x -> function y -> ge_string x y) t; t;;
- : string vect = [|"les "; "copains!"; "Salut "|]
```

```
# let t = [|"Salut "; "les "; "copains!"|] in
  tri (function x -> function y -> le_string x y) t; t;;
- : string vect = [|"Salut "; "copains!"; "les "|]
```

On constate sur cet exemple que les lettres majuscules précèdent les lettres minuscules dans l'ordre lexicographique utilisé en Caml. Nous pouvons maintenant appliquer partiellement la procédure `tri` à des ordres habituels, pour obtenir des procédures de tri spécialisées :

```
# let tri_croissant t = tri (function x -> function y -> x <= y) t;;
tri_croissant : 'a vect -> unit = <fun>
# let tri_décroissant t = tri (function x -> function y -> x >= y) t;;
tri_décroissant : 'a vect -> unit = <fun>
# let tri_du_dictionnaire = tri le_string;;
tri_du_dictionnaire : string vect -> unit = <fun>
# let t = [| "Bonjour"; "tout"; "le"; "monde" |] in
  tri_du_dictionnaire t; t;;
- : string vect = [|"Bonjour"; "le"; "monde"; "tout"|]
```

En conclusion, nous avons pu définir une procédure de tri très générale : grâce au polymorphisme, cette procédure s'applique à tout type de tableau ; grâce à la pleine fonctionnalité, elle s'applique à tout type d'ordre. On retrouve les procédures de tri habituelles en spécialisant certains paramètres de la procédure générale.

4.6 La pleine fonctionnalité

Nous avons vu que les fonctions peuvent être passées en arguments ou rendues en résultat, comme toutes les autres données. Plus étonnant encore, on les manipule comme des valeurs ordinaires à l'intérieur des structures de données. Nous étudions maintenant un exemple qui nous amène très naturellement à utiliser des tableaux de fonctions.

Menu à deux cas

Notre but est d'écrire une fois pour toutes une procédure qui affiche un menu, lit le choix de l'utilisateur et lance l'option correspondante du menu. Pour simplifier, nous nous restreignons dans un premier temps aux menus qui offrent exactement deux possibilités. La procédure prend donc quatre arguments : deux messages d'invite à afficher et deux procédures correspondantes. Après avoir affiché le menu, elle lit l'option retenue par l'utilisateur à l'aide de la fonction prédéfinie `read_int`, qui lit un entier tapé au clavier, puis appelle l'option correspondante, en testant la réponse de l'utilisateur.

```
# let menu invite1 option1 invite2 option2 =
    print_string ("<0>: " ^ invite1); print_string " ";
    print_string ("<1>: " ^ invite2); print_newline ();
    print_string "Choisissez votre option: ";
    let réponse = read_int () in
    if réponse = 0 then option1 () else option2 ();;
menu : string -> (unit -> 'a) -> string -> (unit -> 'a) -> 'a = <fun>
```

Pour nos essais, nous définissons deux petites procédures qui impriment simplement un message au terminal :

```
# let au_revoir () = print_string "Au revoir"; print_newline ();;
au_revoir : unit -> unit = <fun>
# let continuer () = print_string "Continuons!"; print_newline ();;
continuer : unit -> unit = <fun>
```

Nous obtenons alors le dialogue suivant :

```
# menu "Arrêter" au_revoir
       "Continuer" continuer;;
<0>: Arrêter <1>: Continuer
Choisissez votre option: 1
Continuons!
- : unit = ()
```

Menu à plusieurs cas

Pour généraliser la procédure précédente à un nombre quelconque d'options, il suffit de lui passer deux tableaux en arguments : un tableau de chaînes de caractères pour les messages d'invite et un tableau de procédures pour les options. Il faut maintenant écrire le menu avec une boucle `for` parcourant le tableau des messages, puis lire l'option choisie par l'utilisateur et sélectionner la procédure correspondante du tableau des options.

```
# let menu invites options =
    for i = 0 to vect_length invites - 1 do
      print_string
        ("<" ^ (string_of_int i) ^ ">: " ^ invites.(i) ^ " ")
    done;
    print_newline ();
    print_string "Choisissez votre option: ";
    let réponse = read_int () in
    options.(réponse) ();;
menu : string vect -> (unit -> 'a) vect -> 'a = <fun>
```

La fonction prédéfinie `string_of_int` renvoie la chaîne de caractères correspondant à son argument entier.

À titre démonstratif, nous appelons la procédure avec une troisième option qui consiste à ne rien faire : la procédure associée est simplement la fonction identité (spécialisée au type `unit`) que nous fournissons comme une fonction anonyme.

```
# menu [| "Arrêter"; "Continuer"; "Ne rien faire" |]
       [| au_revoir; continuer; (function () -> ()) |];;
<0>: Arrêter <1>: Continuer <2>: Ne rien faire
Choisissez votre option 2
- : unit = ()
```

Utiliser les types pour éviter les erreurs

Fournir deux tableaux distincts pour les options et les messages d'invite est source d'erreurs, puisque le typage n'assure pas la correspondance entre l'invite et l'option. La correction est aisée : il suffit de n'utiliser qu'un seul tableau contenant des paires dont

le premier élément est un message d'invite et le second l'option associée. Cet exemple nous amène à définir les fonctions d'accès aux composantes d'une paire, traditionnellement nommées `fst` (pour *first*, qui signifie « premier » en anglais) et `snd` (pour *second*, « second »). Bien que ces fonctions soient prédéfinies en Caml, nous écrivons leur code car il est élégant. On opère tout simplement par filtrage de la paire argument :

```
# let fst (x, y) = x;;
fst : 'a * 'b -> 'a = <fun>
# let snd (x, y) = y;;
snd : 'a * 'b -> 'b = <fun>
```

Une fois de plus, le polymorphisme nous autorise à définir ces deux fonctions pour tous les types de paires. La fonction `menu` est maintenant sans surprises.

```
# let menu invites_options =
    for i = 0 to vect_length invites_options - 1 do
      print_string ("<" ^ (string_of_int i) ^ ">: ");
      print_string (fst (invites_options.(i)) ^ " ")
    done;
    print_newline ();
    print_string "Choisissez votre option: ";
    let réponse = read_int () in
    (snd (invites_options.(réponse))) ();;
menu : (string * (unit -> 'a)) vect -> 'a = <fun>

# menu [| ("Arrêter", au_revoir);
          ("Continuer", continuer);
          ("Ne rien faire", (function () -> ())) |];;
<0>: Arrêter <1>: Continuer <2>: Ne rien faire
Choisissez votre option: 0
Au revoir
- : unit = ()
```

Un menu polymorphe très général

Réfléchissons encore un peu sur la procédure `menu` : la quintessence de cette procédure n'est pas d'appliquer directement les options, mais plutôt de retourner un certain élément d'un tableau d'options, selon la réaction de l'utilisateur aux propositions affichées. Un pas de plus dans la généralisation consiste donc à ne pas considérer que les options doivent forcément être des procédures. On se contente alors de retourner le deuxième élément du couple correspondant au message d'invite choisi par l'utilisateur.

```
# let menu invites_options =
    for i = 0 to vect_length invites_options - 1 do
      print_string ("<" ^ (string_of_int i) ^ ">: ");
      print_string (fst (invites_options.(i)) ^ " ");
      print_string " "
    done;
    print_newline ();
    print_string "Choisissez votre option";
    let réponse = read_int () in
    snd (invites_options.(réponse));;
```

```
menu : (string * 'a) vect -> 'a = <fun>
```

Ainsi, la procédure `menu` retourne aussi bien des entiers que des fonctions. Voici par exemple un morceau de programme qui déterminerait le niveau de difficulté à prendre en compte dans un jeu. Ici la fonction `menu` retourne un entier.

```
# let niveau_de_difficulté =
     print_string "Êtes-vous"; print_newline ();
     menu [| ("Débutant ?", 1);
             ("Amateur ?", 2);
             ("Amateur confirmé ?", 5);
             ("Expert ?", 10) |];;
Êtes-vous
<0>: Débutant ? <1>: Amateur ? <2>: Amateur confirmé ? <3>: Expert ?
Choisissez votre option: 0
niveau_de_difficulté : int = 1
```

Nous avons cependant toujours le loisir d'appeler `menu` avec des options fonctionnelles.

```
# let option =
     menu [| ("Arrêter", au_revoir);
             ("Continuer", continuer);
             ("Ne rien faire", (function () -> ())) |] in
  option ();;
<0>: Arrêter <1>: Continuer <2>: Ne rien faire
Choisissez votre option: 0
Au revoir
- : unit = ()
```

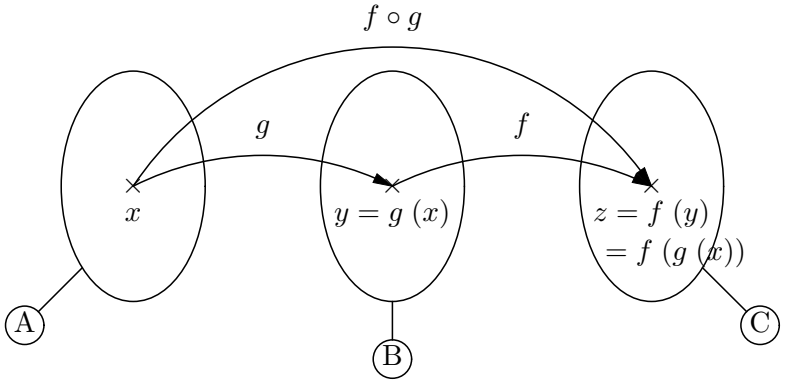
Il est bien entendu que la fonction `menu` reste naïve : il lui faudrait tester la validité de la réponse de l'utilisateur et l'interroger à nouveau en cas d'erreur. La validation de la réponse pourrait s'effectuer à l'aide d'une fonction, argument supplémentaire de `menu`. On peut aussi envisager de lire des chaînes de caractères au lieu de nombres (par exemple `"oui"` ou `"non"`). Il n'en demeure pas moins que le polymorphisme et la pleine fonctionnalité nous permettent d'écrire une fonction très générale dans laquelle les problèmes de mise en page des menus, d'obtention d'une réponse et de validation de la réponse obtenue seront factorisés une fois pour toutes.

Vous en savez maintenant assez pour passer au chapitre suivant. Ce qui suit est étonnant mais technique. En particulier, nous verrons que le langage est assez puissant pour définir un moyen automatique de passer de la version curryfiée à la version non curryfiée d'une fonction.

4.7 Composition de fonctions

En utilisant des fonctionnelles, on parvient à programmer des notions mathématiques qui paraissent *a priori* hors de portée d'une machine. Pour les lecteurs férus de mathématiques, nous allons étudier un exemple surprenant : la composition des fonctions. Il est non seulement possible de définir la composition de deux fonctions données en Caml, mais même d'écrire un programme qui implémente le fameux opérateur « $\circ$ ».

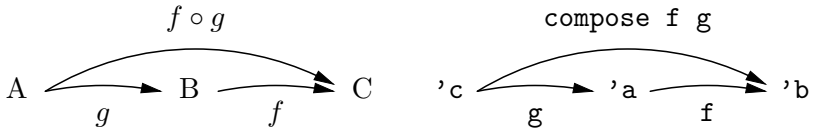
Rappelons que composer deux fonctions revient à les appliquer successivement : la composée des fonctions f et g, qu'on note $f \circ g$ en mathématiques (prononcer « f rond g »), est la fonction h définie par $h(x) = f(g(x))$. On fait souvent le schéma suivant :



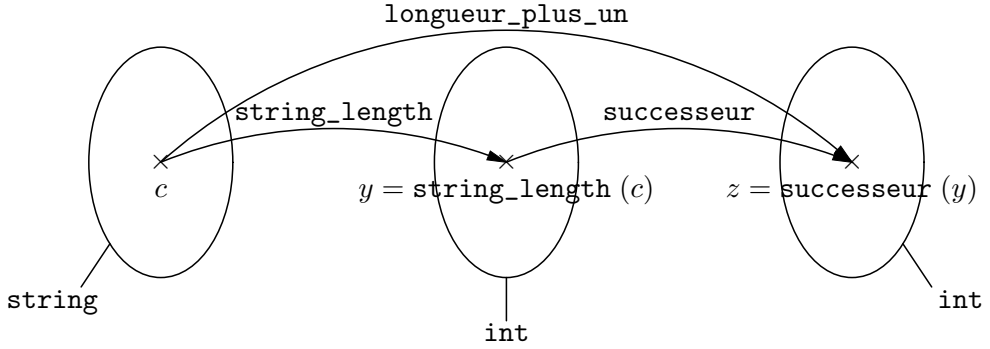
Ainsi, l'opérateur ∘ des mathématiques est une fonctionnelle qui prend deux fonctions en arguments et renvoie une fonction : leur composée. Il n'y a pas de difficultés à définir l'opérateur ∘ en Caml ; nous l'implémentons sous le nom de `compose`.

```
# let compose f g = function x -> f (g x);;
compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Le type de la fonction `compose` reflète fidèlement les restrictions qu'on doit imposer à deux fonctions pour pouvoir effectivement les composer. D'ailleurs, ces restrictions apparaissent dans notre schéma : il faut que l'ensemble de départ de la fonction f soit le même que l'ensemble d'arrivée de la fonction g. De plus, par définition, la composée $f \circ g$ de f et de g a pour ensemble de départ celui de g et pour ensemble d'arrivée celui de f. On le constate graphiquement, si l'on ne fait figurer dans le schéma de la composition que les ensembles et les fonctions qui interviennent (schéma ci-dessous à gauche). Le contrôleur de type de Caml calcule un type où les noms A, B, C sont remplacés respectivement par les paramètres `'c`, `'a` et `'b` (schéma de droite).



Le contrôleur de type a donc retrouvé tout seul les contraintes mathématiques et les vérifiera automatiquement à chaque composition de fonctions. Considérons l'exemple d'école du successeur de la longueur d'une chaîne de caractères, d'abord sous forme graphique, puis en Caml.



```
# let longueur_plus_un = compose successeur string_length;;
longueur_plus_un : string -> int = <fun>
# longueur_plus_un "OK";;
- : int = 3
```

La composition de ces deux fonctions dans l'ordre inverse n'a pas de sens, ce que le contrôleur de types signale :

```
# compose string_length successeur;;
Entrée interactive:
>compose string_length successeur;;
>                      ^^^^^^^^^^
Cette expression est de type int -> int,
mais est utilisée avec le type int -> string.
```

La fonctionnelle de curryfication

Nous allons définir une fonctionnelle, `curry`, pour obtenir automatiquement la version curryfiée d'une fonction non curryfiée à deux arguments. Notre fonctionnelle prendra donc en argument une fonction `f` dont l'argument est une paire `(x, y)` et rendra en résultat une fonction à deux arguments (qui est donc de la forme `function x -> function y ->` ...) rendant le même résultat que `f`. On obtient donc

```
let curry f =
    (function x -> function y -> « même résultat que f pour x et y »);;
```

Puisque le « même résultat que `f` pour `x` et `y` » n'est autre que `f (x, y)`, on a donc simplement :

```
# let curry f = function x -> (function y -> f (x, y));;
curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
```

Le type de la fonction `curry` est plus clair si l'on rétablit les parenthèses omises par l'imprimeur de types de Caml (une fois n'est pas coutume). En effet `('a * 'b -> 'c) -> 'a -> 'b -> 'c` se lit aussi `('a * 'b -> 'c) -> ('a -> 'b -> 'c)`.

La fonctionnelle de décurryfication

La fonctionnelle réciproque, `uncurry`, n'est pas plus complexe. Elle prend en argument une fonction curryfiée `g` et rend en résultat une fonction ayant un seul argument qui est une paire. Ce résultat est donc une valeur de la forme `function (x, y) ->` ...

```
let uncurry g =
    (function (x, y) -> « même résultat que g pour x et y »);;
```

Or, puisque `g` est curryfiée à deux arguments, le « même résultat que `g` pour `x` et `y` » est `g x y`.

```
# let uncurry g = function (x, y) -> g x y;;
uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

De même que pour `curry`, le type de `uncurry` se lit plus facilement en rétablissant les parenthèses autour de `'a * 'b -> 'c`.

Leurs composées

Intuitivement, il est clair que les fonctionnelles `curry` et `uncurry` sont réciproques l'une de l'autre : à partir d'une fonction à « un argument de type paire » `curry` renvoie une fonction « à deux arguments », tandis que `uncurry` fait l'inverse. D'ailleurs :

```
# let id_curry f = (compose curry uncurry) f;;
id_curry : ('a -> 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
# let id_uncurry f = (compose uncurry curry) f;;
id_uncurry : ('a * 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

Une fonctionnelle inutile

Considérez la fonctionnelle suivante qui applique une fonction `f` donnée à un argument `x` donné :

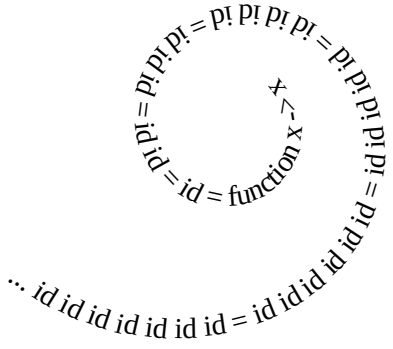
```
# let applique f x = f x;;
applique : ('a -> 'b) -> 'a -> 'b = <fun>
```

Cette fonctionnelle ne sert à rien ! D'abord, si vous examinez soigneusement son type, vous vous rendrez compte que c'est une spécialisation du type de l'identité : `('a -> 'b) -> 'a -> 'b` est une abréviation pour `('a -> 'b) -> ('a -> 'b)`, c'est-à-dire `'a -> 'a` avec `'a` spécialisé en `('a -> 'b)`. Donc, `applique` pourrait bien ne rien faire, comme la fonction `identité`. On le démontre facilement, en utilisant toujours la règle η avec l'argument x cette fois :

`let applique f x = f x;;` équivaut à `let applique f = f;;`

ce qui démontre que `applique` est sémantiquement équivalente à l'identité, spécialisée aux fonctions. En effet, `applique` impose à son argument `f` d'être fonctionnel puisqu'elle l'applique à l'argument `x`. Cela nous fournit un exemple où l'application de la règle η change le type de la fonction qu'on définit, le rendant plus général :

```
# let applique f = f;;
applique : 'a -> 'a = <fun>
```



5
Listes

La tête et la queue, sans les oreilles ...

ARMI LES STRUCTURES DE DONNÉES PRÉDÉFINIES en Caml, les listes ont un statut privilégié : elles sont d'emploi fréquent, car simples à manipuler et très générales. Si vous savez que les listes peuvent à peu près tout coder, mais qu'il ne faut pas pour autant les mettre à toutes les sauces, en bref, si vous connaissez bien `it_list` et que justement vous évitez de l'employer trop souvent, alors lisez directement le chapitre suivant.

Nous verrons, dans celui-ci, une autre représentation des polynômes par des listes de paires d'entiers et nous animerons le jeu des tours de Hanoi en montrant l'évolution de l'état du jeu. Au passage, nous approfondirons le filtrage (appel explicite au filtrage, synonymes dans les filtres, filtrage multiple) et nous verrons des exemples de fonctionnelles sur les listes.

5.1 Présentation

Syntaxe

Les listes sont des suites homogènes de valeurs, entourées de crochets `[` et `]`. Comme pour les tableaux, les éléments des listes sont séparés par un point-virgule « `;` ».

```
# [1; 2; 3];;
- : int list = [1; 2; 3]
```

Au contraire des tableaux, on n'accède pas directement à un élément de liste : il faut parcourir séquentiellement la liste pour atteindre l'élément recherché. En revanche, les listes peuvent grossir dynamiquement alors que les tableaux ont une taille déterminée, fixée lors de leur construction. Cependant on ajoute toujours les éléments au début d'une liste et non à la fin. C'est contraire à la pratique courante : quand on tient à jour une liste de courses, on ajoute généralement la prochaine course à faire à la fin de sa liste ...

Toutes les listes sont construites avec les deux constructeurs de listes, « `[]` » (qu'on prononce « nil », d'après l'anglais *nil* qui signifie néant et qui provient du latin *nihil* qui veut dire rien) et « `::` » (qu'on prononce « conse », par abréviation de « constructeur de

liste »). `[]` est la liste vide et `::` est l'opérateur infixe qui ajoute un élément en tête d'une liste. Tout comme le tableau vide, la liste vide est polymorphe.

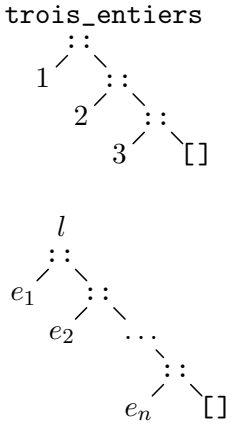
```
# [];;
- : 'a list = []
# 0 :: [1; 2; 3];;
- : int list = [0; 1; 2; 3]
# 3 :: [];;
- : int list = [3]
# 1 :: 2 :: 3 :: [];;
- : int list = [1; 2; 3]
```

Représentation graphique des listes

Pour expliquer certaines fonctions qui manipulent des listes, nous représentons graphiquement les calculs qu'elles effectuent. Pour cela nous dessinons les listes comme des peignes dont les dents contiennent les éléments de la liste. Par exemple, la liste `trois_entiers` définie par :

```
# let trois_entiers = [1; 2; 3];;
trois_entiers : int list = [1; 2; 3]
```

est représentée par le schéma ci-contre. Cette représentation en peigne a l'avantage de mettre en évidence la construction de la liste à partir de ses éléments, de la liste vide, et d'applications successives du constructeur « `::` ». En effet, la liste `trois_entiers` vaut `1 :: 2 :: 3 :: []` et vous aurez sans doute remarqué que la notation $[e_1;\ e_2;\ \ldots\ ;\ e_n]$ est une abréviation pour e_1 `::` e_2 `::` ... `::` e_n `::` `[]`. Lorsque nous expliquerons des fonctions sur les listes, la liste argument la plus générale sera notée l et ses éléments seront notés e_1, e_2, ... , e_n. Cette liste la plus générale sera donc décrite graphiquement par le peigne en marge.



Filtrage des listes

Le filtrage est étendu aux listes, si bien qu'on teste si une liste est vide avec la fonction suivante :

```
# let nulle = function
    | [] -> true
    | _  -> false;;
nulle : 'a list -> bool = <fun>
```

Ce texte Caml se lit comme suit : si l'argument de la fonction `nulle` est la liste vide, alors retourner `true` (`[] -> true`) ; dans tous les autres cas (`_`), retourner `false`. Par exemple :

```
# (nulle []), (nulle [1]);;
- : bool * bool = true, false
```

De plus, il est possible de nommer la tête ou le reste d'une liste argument avec des filtres utilisant « `::` ».

```
# let tête = function
    | t :: r -> t
    | _ -> failwith "tête";;
tête : 'a list -> 'a = <fun>
# tête [1; 2; 3; 4];;
- : int = 1
```

La clause `t :: r -> t` signifie : si la liste argument n'est pas vide et qu'on appelle sa tête `t` et son reste `r` (`t :: r`), alors retourner `t`. La clause `| _ ->` signifie comme d'habitude « dans tous les autres cas ». La fonction `failwith` est une fonction prédéfinie qui signale une erreur : en anglais, `failwith` signifie « échouer avec ». Vous devinez donc que `failwith "tête"` signale une erreur dans la fonction `tête`.

```
# tête [];;
Exception non rattrapée: Failure "tête"
```

Le système nous signale ainsi une exception non rattrapée (« Uncaught exception »), c'est-à-dire un échec, avec pour message la chaîne `tête` (`Failure "tête"`). Nous reviendrons sur ce mécanisme d'échec et sur la notion d'exception au chapitre 7.

Comme d'habitude, il n'est pas obligatoire de nommer une partie de la valeur argument qu'on n'utilise pas. Par exemple, il n'est pas nécessaire de nommer la tête d'une liste pour retourner son reste (on dit aussi la queue de la liste) :

```
# let reste = function
    | _ :: r -> r
    | _ -> failwith "reste";;
reste : 'a list -> 'a list = <fun>
# reste [1; 2; 3; 4];;
- : int list = [2; 3; 4]
```

5.2 Programmation assistée par filtrage

Il est facile de définir une fonction qui additionne tous les éléments d'une liste d'entiers : si la liste est vide, alors le résultat est 0 ; sinon, on ajoute le premier élément de la liste à la somme des éléments du reste de la liste.

```
# let rec somme = function
    | [] -> 0
    | x :: l -> x + somme l;;
somme : int list -> int = <fun>
```

On calcule le produit des éléments d'une liste de façon similaire.

```
# let rec produit = function
    | [] -> 1
    | x :: l -> x * produit l;;
produit : int list -> int = <fun>
```

Ces deux dernières fonctions sont des exemples représentatifs : les fonctions sur les listes sont le plus souvent récursives et opèrent un filtrage sur la liste argument.

Généralement, la fonction s'arrête quand elle rencontre une liste vide et s'appelle récursivement quand la liste n'est pas vide. Le squelette général d'une telle fonction est donc :

```
let rec f = function
  | [] -> « valeur de base »
  | x :: l -> ... f(l) ...;;
```

Cela vous explique pourquoi les fonctions `nulle`, `tête` et `reste` ne sont guère employées en Caml : à quoi bon tester explicitement `if nulle(l) then` ... `else` ..., quand le filtrage permet de le faire bien plus élégamment. En effet, en n'employant pas le filtrage, le squelette général d'une fonction récursive sur les listes s'écrirait bien plus lourdement :

```
let rec f l =
    if nulle l
    then « valeur de base »
    else let x = tête l and l' = reste l in ... f(l') ...;;
```

Le modèle général de fonction récursive définie par filtrage sur les listes est à rapprocher de celui qu'on a vu sur les entiers :

```
let rec f = function
  | 0 -> « solution simple »
  | n -> ... f (n - 1) ...;;
```

Schématiquement, le squelette le plus général de fonctions récursives Caml est suggéré par le pseudo-programme suivant :

```
let rec f = function
  | « cas simple » -> « solution simple »
  | « autre cas simple » -> « autre solution simple »
  ⋮
  | « cas général » -> ... f (« cas plus simple ») ...;;
```

Il est donc très fréquent que le filtrage dirige l'écriture des fonctions. Par exemple, si l'on définit une fonction sur les listes, on écrira (presque) sans réfléchir :

```
let rec f = function
  | [] ->
  | x :: l ->
```

À titre d'entraînement avant d'attaquer les fonctionnelles d'usage général sur les listes, nous allons mettre en pratique cette « *programmation assistée par filtrage* » dans un cas simple : le tri des listes.

5.3 Tri par insertion

Nous implémentons le tri par insertion, qui est un algorithme naturellement récursif. On suppose qu'une sous-partie du tableau à trier est déjà triée et on y insère à la bonne place le prochain élément de la partie non triée du tableau. Nous en donnons une version fonctionnelle sur les listes.

Tri sur listes

L'idée est qu'il est facile de ranger un élément à sa place dans une liste d'éléments déjà triée. Nous supposons donc avoir déjà écrit une fonction `insère` qui insère un élément à la bonne place dans une liste triée. Nous écrivons maintenant la fonction de tri. Cette fonction travaille sur des listes ; elle doit donc envisager les deux cas possibles de listes :

```
let tri_par_insertion = function
  | [] -> ...
  | x :: reste -> ... ;;
```

Le cas de la liste vide est simple : une liste vide est évidemment triée ; on renvoie donc la liste vide.

```
let tri_par_insertion = function
  | [] -> []
  | x :: reste -> ... ;;
```

Dans l'autre cas, on va commencer par trier le reste de la liste. C'est déjà possible, bien que nous n'ayons pas encore écrit notre fonction de tri : il suffit d'appeler récursivement la fonction `tri_par_insertion` que nous sommes justement en train d'écrire ...

```
let rec tri_par_insertion = function
  | [] -> []
  | x :: reste -> ... tri_par_insertion reste;;
```

Il nous suffit maintenant de mettre l'élément `x` à la bonne place dans le reste maintenant trié de la liste. C'est facile : on se contente d'appeler la fonction `insère`. Nous obtenons :

```
let rec tri_par_insertion = function
  | [] -> []
  | x :: reste -> insère x (tri_par_insertion reste);;
```

La fonction de tri est terminée. Il nous reste à écrire la fonction `insère`. Par le même raisonnement que ci-dessus on commence par en écrire le squelette :

```
let insère élément = function
  | [] -> ...
  | x :: reste -> ...;;
```

Le cas de la liste vide est encore une fois simple : il suffit de retourner une liste réduite à l'élément qu'on souhaite insérer.

```
let insère élément = function
  | [] -> [élément]
  | x :: reste -> ...;;
```

Dans l'autre cas, la liste où l'on veut insérer `élément` commence par `x`. Si `élément` est plus petit que `x` alors c'est le plus petit de tous les éléments de la liste `x :: reste`, puisque celle-ci est triée par hypothèse. On place donc `élément` au début de la liste `x :: reste`.

```
let insère élément = function
  | [] -> [élément]
  | x :: reste -> if élément <= x then élément :: x :: reste else ...;;
```

Dans le cas contraire, c'est `x` le plus petit élément de la liste résultat ; ce résultat sera donc `x :: ...` Il nous reste à insérer `élément` dans la liste `reste`. Un petit appel récursif `insère élément reste` et le tour est joué :

```
# let rec insère élément = function
    | [] -> [élément]
    | x  ::  reste ->
       if élément <= x then élément :: x :: reste
                       else x :: (insère élément reste);;
insère : 'a -> 'a list -> 'a list = <fun>
```

Il nous reste à définir effectivement la fonction de tri et à l'essayer :

```
# let rec tri_par_insertion = function
    | [] -> []
    | x :: reste -> insère x (tri_par_insertion reste);;
tri_par_insertion : 'a list -> 'a list = <fun>
# tri_par_insertion [3; 2; 1];;
- : int list = [1; 2; 3]
```

Synonymes dans les filtres

Pour améliorer la lisibilité du code de la fonction `insère`, nous introduisons une facilité de nommage supplémentaire dans les filtres.

Il arrive que l'on veuille examiner la forme d'une valeur tout en nommant cette valeur. Considérez la fonction qui rend la valeur absolue d'un monôme, représenté comme une paire d'entier (coefficient, degré) :

```
# let abs_monôme = function
      (a, degré) -> if a < 0 then (-a, degré) else (a, degré);;
abs_monôme : int * 'a -> int * 'a = <fun>
```

Ce code est parfaitement correct, mais dans le cas où le coefficient est positif on aimerait rendre directement le monôme reçu en argument. Le code serait plus clair, puisqu'il n'y aurait pas besoin d'une petite gymnastique mentale pour se rendre compte que l'expression `(a, degré)` correspond exactement au filtre de la clause. Autrement dit, nous voudrions nommer `monôme` le filtre `(a, degré)` et rendre `monôme` quand `a` est positif. Dans ce cas, on introduit le nom choisi avec le mot-clé `as` (qui se prononce « ase » et signifie « en tant que » en anglais).

Synonymes dans les filtres ::= *filtre* `as` *nom*

Nous obtenons :

```
# let abs_monôme = function
      (a, degré) as monôme -> if a < 0 then (-a, degré) else monôme;;
abs_monôme : int * 'a -> int * 'a = <fun>
```

Maintenant le nommage indique à l'évidence qu'aucune transformation n'est faite sur le monôme, alors que l'expression `(a, degré)`, bien qu'équivalente, cache un peu qu'elle n'est autre que l'argument de la fonction.

Pour la fonction `insère`, l'usage d'un filtre synonyme pour nommer la liste argument clarifie également un peu le code :

```
# let rec insère élément = function
    | [] -> [élément]
    | x :: reste as l ->
       if élément <= x then élément :: l
                       else x :: (insère élément reste);;
insère : 'a -> 'a list -> 'a list = <fun>
```

Généralisation du tri à tout type d'ordre

Pour généraliser la fonction de tri à toute sorte d'ordres, il suffit de passer la fonction de comparaison en argument, comme on l'a vu au chapitre 4. Les fonctions `insère` et `tri_par_insertion` prennent alors un argument supplémentaire, `ordre`, qu'on utilise pour comparer les éléments, à la place de la comparaison `<=`.

```
# let rec insère ordre élément = function
    | [] -> [élément]
    | x :: reste as l ->
       if ordre élément x then élément :: l
       else x :: (insère ordre élément reste);;
insère : ('a -> 'a -> bool) -> 'a -> 'a list -> 'a list = <fun>

# let rec tri_par_insertion ordre = function
    | [] -> []
    | x :: reste -> insère ordre x (tri_par_insertion ordre reste);;
tri_par_insertion : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>
```

La même fonction nous permet maintenant de trier indifféremment des listes de chaînes ou de nombres, à l'endroit ou à l'envers :

```
# tri_par_insertion (function x -> function y -> x <= y) [3; 1; 2];;
- : int list = [1; 2; 3]
# tri_par_insertion (function x -> function y -> x >= y) [3; 1; 2];;
- : int list = [3; 2; 1]
# tri_par_insertion (function x -> function y -> ge_string x y)
                    ["Salut "; "les "; "copains!"];;
- : string list = ["les "; "copains!"; "Salut "]
# tri_par_insertion (function x -> function y -> le_string x y)
                    ["Salut "; "les "; "copains!"];;
- : string list = ["Salut "; "copains!"; "les "]
```

Remarque de complexité : on démontre que ce tri est *quadratique* ($O(n^2)$) en moyenne (sur un jeu de données tirées au hasard). Dans le pire des cas, c'est-à-dire quand le jeu de données nécessite le plus d'opérations (ce qui correspond pour ce tri à une liste triée en ordre inverse), le tri par insertion est également quadratique. En revanche, il est linéaire pour une liste déjà triée.

5.4 Fonctionnelles simples sur les listes

Les listes, comme toutes les structures de données polymorphes, se prêtent naturellement à la définition de fonctionnelles réutilisables dans de nombreuses situations. Nous présentons dans cette section quelques-unes de ces fonctionnelles, parmi les plus simples ; nous en verrons d'autres, plus complexes, à la section 5.9.

Faire une action sur les éléments d'une liste

Étant données une fonction f et une liste l, la fonctionnelle `do_list` applique f tour à tour à tous les éléments de l. Cela correspond donc à effectuer des appels à f en séquence sur les éléments de l. Autrement dit, évaluer `do_list` f $[e_1; e_2; \ldots ; e_n]$ signifie exécuter la séquence `begin` $f\ e_1; f\ e_2; \ \ldots\ ; f\ e_n$; `() end`. Par exemple :

```
# do_list print_int [1; 2; 3];;
123- : unit = ()
# do_list (function i -> print_int i; print_char ' ') [1; 2; 3];;
1 2 3 - : unit = ()
```

Écrivons `do_list` :

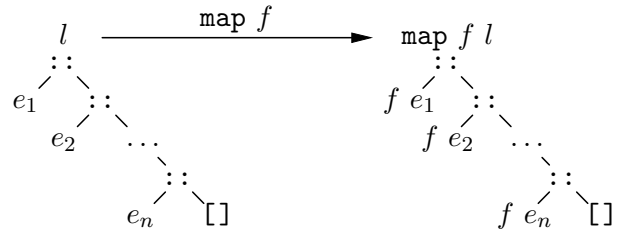
```
# let rec do_list f = function
    | [] -> ()
    | x :: l -> f x; do_list f l;;
do_list : ('a -> 'b) -> 'a list -> unit = <fun>
```

Le raisonnement est le suivant : si la liste argument est vide, il n'y a rien à faire. Sinon, la liste argument est de la forme `x :: l`, car elle n'est pas vide. En ce cas, il faut évaluer en séquence `f(x)`, puis l'action de `f` sur le reste de la liste, qu'on obtient par un appel récursif à `do_list`.

Remarquons que le type de la fonction `do_list` indique clairement que la fonction f doit accepter en argument les éléments de la liste, puisque f a pour type `'a -> 'b` et que la liste a pour type `'a list`. En revanche, les résultats des appels à f sont ignorés : f peut rendre un résultat de n'importe quel type.

Appliquer à tous les éléments d'une liste

Étant données une fonction f et une liste l, la fonctionnelle `map` retourne la liste des résultats obtenus en appliquant f à chaque élément de l. Par exemple, `map successeur [1;2;3]` retourne la liste `[2;3;4]`, tandis que `map carré [1;2;3]` retourne `[1;4;9]`. L'expression `map` f $[e_1; e_2; \ldots ; e_n]$ retourne donc la liste $[f\ e_1; f\ e_2; \ \ldots\ ; f\ e_n]$. Graphiquement, la fonction `map` « distribue » f sur les éléments de la liste argument :



Le nom `map` provient du langage Lisp et signifie « application multiple ». La fonction s'écrit très simplement : si la liste argument est vide le résultat est la liste vide. Sinon, la liste argument est de la forme `x :: l` et il suffit de mettre en tête du résultat l'application de `f` à `x`, le reste du résultat étant fourni par un appel récursif à `map`.

```
# let rec map f = function
    | [] -> []
    | x :: l -> f x :: map f l;;
map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Cette fois-ci le type d'arrivée de la fonction `f` n'est plus ignoré : il doit être identique au type des éléments de la liste des résultats.

```
# map succ [1; 2; 3];;
- : int list = [2; 3; 4]
```

```
# map string_length ["Bonjour"; "tout"; "le "; "monde!"];;
- : int list = [7; 4; 3; 6]
```

Nous allons utiliser nos nouvelles connaissances sur les listes pour calculer des polynômes qui nous étaient précédemment inaccessibles.

5.5 Les polynômes creux

Une représentation affinée

Notre premier programme de manipulation des polynômes (section 3.3) représentait les polynômes par des tableaux d'entiers. Cette représentation est très dispendieuse quand les polynômes ont peu de monômes dont les degrés sont très différents : il serait insupportable de représenter le polynôme $1 + x^{10000}$ par un tableau à 10001 cases dont 9999 comporteraient des zéros ! Les polynômes ayant cette propriété d'avoir une forte proportion de coefficients nuls sont appelés polynômes *creux*. Par opposition nous avions appelé polynômes *pleins* les polynômes de la section 3.3, représentés par le tableau des coefficients de leurs monômes. Une représentation plus compacte des polynômes creux utilise tout naturellement la liste ordonnée des coefficients non nuls, avec leurs degrés respectifs. Il s'agit donc de listes de paires d'entiers. Dans cette représentation, le polynôme $1 + 2X^3$ serait :

```
# let p = [(1, 0); (2, 3)];;
p : (int * int) list = [1, 0; 2, 3]
```

Nous allons redéfinir les opérations sur les polynômes, en travaillant sur des polynômes creux.

Impression des polynômes creux

L'impression est très simple : nous utilisons exactement la même fonction d'impression des monômes que dans le cas des polynômes pleins, mais nous l'itérons sur la liste des coefficients à l'aide de la fonctionnelle `do_list`.

```
# let imprime_polynôme_creux p =
      do_list (function (a, degré) -> imprime_monôme a degré) p;;
imprime_polynôme_creux : (int * int) list -> unit = <fun>

# imprime_polynôme_creux p;;
1 + 2x^3- : unit = ()
```

(Les amateurs de casse-tête compareront la fonction `imprime_polynôme_creux` avec la fonction `do_list (uncurry imprime_monôme)` ou même `compose do_list uncurry imprime_monôme`.)

L'addition est un peu plus compliquée. En effet, elle opère un filtrage en parallèle sur les deux polynômes arguments. Ce filtrage est complexe et utilise des traits nouveaux pour nous. Nous les détaillons avant d'analyser le code de l'addition des polynômes creux.

5.6 Filtrage explicite

La construction `match ... with`

Pour filtrer ses deux arguments en parallèle, la fonction d'addition des polynômes utilise un appel explicite au filtrage, mécanisme introduit par le mot-clé `match` :

Appel explicite au filtrage ::= `match` *expression*
`with` *filtrage*

Cette construction a la sémantique d'une conditionnelle par cas, ou encore d'un `if` généralisé travaillant par filtrage. Redéfinissons la fonction `nulle` avec un appel explicite au filtrage : on demande explicitement d'examiner la forme de la liste `l` argument de `nulle` et l'on renvoie dans chaque cas l'expression adéquate.

```
# let nulle l =
      match l with
      | [] -> true
      | _ -> false;;
nulle : 'a list -> bool = <fun>
```

On lit ce filtrage comme la phrase : si la liste `l` est vide, renvoyer `true` ; dans tous les autres cas, renvoyer `false`.

En utilisant le filtrage explicite, nous pouvons très facilement écrire la fonction qui concatène deux listes : si la première liste est vide, le résultat est la deuxième ; sinon, il faut mettre la tête de la première liste devant la concaténation du reste de cette liste avec la deuxième liste. Ce qui, parce que bien conçu, s'énonce clairement ainsi :

```
# let rec concatène l1 l2 =
      match l1 with
      | [] -> l2
      | x :: reste -> x :: concatène reste l2;;
concatène : 'a list -> 'a list -> 'a list = <fun>
# concatène [1; 2; 3] [4; 5; 6];;
- : int list = [1; 2; 3; 4; 5; 6]
```

Filtrage simultané de deux valeurs

Pour envisager tous les cas concernant la forme de deux listes, il suffit de filtrer explicitement la paire des deux listes. Ainsi, la fonction suivante détermine si deux listes ont même longueur :

```
# let rec même_longueur l1 l2 =
      match (l1, l2) with
      | ([], []) -> true
      | (_ :: reste1, _ :: reste2) -> même_longueur reste1 reste2
      | (_, _) -> false;;
même_longueur : 'a list -> 'b list -> bool = <fun>

# même_longueur [1] [2];;
- : bool = true

# même_longueur [1] [1; 2];;
- : bool = false
```

Le filtre `(_ :: reste1, _ :: reste2)` signifie : si la première liste `l1` n'est pas vide et donc de la forme `_ :: reste1` et si *simultanément* la seconde liste `l2` est non vide et de la forme `_ :: reste2`, alors ... En particulier, les soulignés « `_` » mettent bien en évidence qu'on ne s'intéresse pas à la tête des listes, mais qu'on a nommé leurs restes respectifs `reste1` et `reste2`. Remarquez que le dernier cas du filtrage correspond à des listes d'inégales longueurs : l'une est épuisée mais l'autre ne l'est pas. En effet, si les deux étaient épuisées le premier cas s'appliquerait, tandis que si aucune n'était épuisée, c'est le second qui s'appliquerait. Autrement dit : le cas `(_, _)` regroupe en une seule clause les deux cas `([], _ :: _) -> false` et `(_ :: _, []) -> false`.

Remarquons également que le filtrage simultané ne nécessite pas les parenthèses des couples ; la construction permet de filtrer des expressions séparées par des virgules avec leurs filtres respectifs, eux aussi séparés par des virgules :

```
match l1, l2 with
| [], [] -> true
| _ :: reste1, _ :: reste2 -> ...
```

5.7 Opérations sur les polynômes creux

Addition des polynômes creux

L'addition des polynômes creux va analyser récursivement ses deux arguments pour en construire la somme. Comme nous l'avons vu dans le cas des polynômes pleins, il arrive que certains termes d'un des deux polynômes arguments n'aient pas de correspondants dans l'autre, parce que les polynômes n'ont pas le même degré. Dans le cas des polynômes creux, l'une des listes de monômes sera épuisée avant l'autre. Si nous atteignons ainsi la fin de l'un des polynômes, l'autre constitue le résultat cherché : par exemple, si l'on ajoute un polynôme P_0 réduit à une constante à un autre polynôme P, il faut ajouter les deux monômes de degré 0 de P et P_0, mais le résultat comprend aussi la liste des monômes restants de P. En effet, les coefficients manquants du polynôme de plus bas degré correspondent à des zéros implicites. Dans le cas général, nous ajoutons les termes de même degré ou recopions dans le résultat final les termes qui n'ont pas d'analogue dans l'autre polynôme.

```
# let rec ajoute_polynômes_creux p1 p2 =
    match p1, p2 with
    | _, [] -> p1
    | [], _ -> p2
    | (a1, degré1 as m1) :: reste1, (a2, degré2 as m2) :: reste2 ->
       if degré1 = degré2
       then ((a1 + a2), degré1) :: ajoute_polynômes_creux reste1 reste2
       else if degré1 < degré2
            then m1 :: ajoute_polynômes_creux reste1 p2
            else m2 :: ajoute_polynômes_creux p1 reste2;;
ajoute_polynômes_creux :
 (int * 'a) list -> (int * 'a) list -> (int * 'a) list = <fun>
```

Le filtre `(a1,degré1 as m1) :: reste1, (a2,degré2 as m2) :: reste2` est complexe et nécessite une explication. Il est clairement constitué de deux filtres analogues

séparés par une virgule, l'un pour filtrer `p1` et l'autre pour filtrer `p2`. Examinons celui qui concerne `p1`. Le filtre `(a1,degré1 as m1) :: reste1` signifie que :

- `p1` est une liste non vide dont la tête est filtrée par `(a1, degré1 as m1)` et le reste est nommé `reste1`,
- la tête de `p1` est donc un couple dont les composantes sont nommées `a1` et `degré1`,
- le couple lui-même, `(a1, degré1)`, est nommé `m1` grâce au filtre synonyme `as m1`.

Admirons au passage la puissance et l'élégance du mécanisme de filtrage. Remarquez également que les filtres sont essayés dans l'ordre de présentation dans le filtrage. Par exemple, la valeur `([], [])` sera filtrée par le premier filtre, bien qu'elle soit aussi filtrable par le second. À titre d'exemple, nous calculons la somme des polynômes $X^2 + 3X^4$ et $3 + 2X^2 + 5X^{10}$.

```
# imprime_polynôme_creux
   (ajoute_polynômes_creux [(1,2); (3,4)] [(3,0); (2,2); (5,10)]);;
3 + 3x^2 + 3x^4 + 5x^10- : unit = ()
```

Multiplication des polynômes creux

La multiplication opère également par filtrage simultané de ses deux arguments. Dans le cas où l'un des polynômes est épuisé, il n'y a plus de multiplication à faire. En effet, les monômes manquants ont implicitement des coefficients nuls, donc les multiplications produiront toujours des coefficients nuls. En ce cas, le résultat est donc la liste vide. Sinon, on applique simplement la règle habituelle de distributivité de la multiplication par rapport à l'addition. Voyons : soit m_1 le premier monôme de P_1 et $reste_1$ les autres monômes de P_1. On a $P_1 = m_1 + reste_1$, donc $P_1 \times P_2 = m_1 \times P_2 + reste_1 \times P_2$. Si l'on appelle notre fonction `multiplie_polynômes_creux`, alors $reste_1 \times P_2$ correspond à l'appel récursif `multiplie_polynômes_creux reste1 p2`. Quant à l'expression $m_1 \times P_2$, c'est un cas plus simple où l'on multiplie un polynôme par un monôme. Nous le traiterons par la fonction auxiliaire `multiplie_par_monôme_creux`. L'expression $m_1 \times P_2 + reste_1 \times P_2$ s'écrit donc :

```
ajoute_polynômes_creux
  (multiplie_par_monôme_creux m1 p2)
  (multiplie_polynômes_creux reste1 p2)
```

Il reste à définir `multiplie_par_monôme_creux`. Si m_1 est le monôme et P le polynôme, il suffit de multiplier chaque monôme de P par le monôme m_1, ce qui se fait simplement en multipliant les coefficients et en ajoutant les degrés. En résumé :

```
# let multiplie_par_monôme_creux (a1, degré1) p =
      map (function (a, degré) -> (a * a1, degré1 + degré)) p;;
multiplie_par_monôme_creux :
 int * int -> (int * int) list -> (int * int) list = <fun>
# let rec multiplie_polynômes_creux p1 p2 =
    match p1, p2 with
    | (_, []) -> []
    | ([], _) -> []
    | (m1 :: reste1, _) ->
```

```
        ajoute_polynômes_creux
          (multiplie_par_monôme_creux m1 p2)
          (multiplie_polynômes_creux reste1 p2);;
multiplie_polynômes_creux :
 (int * int) list -> (int * int) list -> (int * int) list = <fun>
```

Nous calculons $(1 + X^{10000})^2$ à titre d'exemple :

```
# let p = [(1, 0); (1, 10000)] in
  imprime_polynôme_creux (multiplie_polynômes_creux p p);;
1 + 2x^10000 + x^20000- : unit = ()
```

En première lecture, vous en savez largement assez pour passer dès maintenant au chapitre suivant.

Polymorphisme et règle η

Nous devons signaler ici une petite difficulté qui apparaît lorsqu'on utilise la règle η pour simplifier une définition, lorsque le résultat doit être polymorphe. Supposons que nous définissions le tri par ordre croissant ainsi :

```
# let tri_croissant l =
    tri_par_insertion (function x -> function y -> x <= y) l;;
tri_croissant : 'a list -> 'a list = <fun>
```

On peut espérer simplifier cette définition à l'aide de la règle η, en supprimant l'argument l :

```
# let tri_croissant =
    tri_par_insertion (function x -> function y -> x <= y);;
tri_croissant : '_a list -> '_a list = <fun>
```

On constate alors que la fonction `tri_croissant` n'a plus le même type, et qu'il apparaît dans ce type d'étranges paramètres de type `'_a`.Au contraire des paramètres `'a` qui signifient *pour tout type* `a`, et dénotent donc des types polymorphes, les paramètres `'_a` signifient *pour un certain type* `a` qui sera déterminé par les utilisations ultérieures de la fonction. La fonction `tri_croissant` est donc *monomorphe* :

```
# tri_croissant [3; 2; 1];;
- : int list = [1; 2; 3]
# tri_croissant;;
- : int list -> int list = <fun>
# tri_croissant ["Bonjour"];;
Entrée interactive:
>tri_croissant ["Bonjour"];;
>               ^^^^^^^^^^
Cette expression est de type string list,
mais est utilisée avec le type int list.
```

Le type inconnu `'_a` est devenu le type `int` et la fonction `tri_croissant` est dorénavant de type `int -> int`.

Ce phénomène est dû à la coexistence en Caml du polymorphisme et des structures mutables. Il est expliqué en détails à la fin de ce livre, page 363, lorsque nous aurons vu les mécanismes qui permettent de le comprendre. Retenons pour l'instant que seule les fonctions (et les constantes) sont susceptibles d'être polymorphes, les définitions de

fonctions obtenues par application partielle d'une fonction plus générale sont monomorphes. Nous avons déjà constaté (page 73) que l'application de la règle η peut modifier le type d'une fonction, le rendant plus général ; ici, c'est l'inverse : on passe d'un type polymorphe à un type monomorphe moins général.

5.8 Animation des tours de Hanoi

En guise d'exercice sur les listes, nous définissons un ensemble de fonctions qui manipulent des listes de chaînes pour animer visuellement le jeu des tours de Hanoi. C'est un vrai programme qui utilise des références et la récursivité. Cela reste cependant un exercice : nous n'avons aucun souci d'efficacité ici. Ce genre de programme d'impression élaborée (le *formatage*) est plus du ressort de la modification physique de chaînes de caractères que de la manipulation de listes.

```
# let blancs n = make_string n ' ';;
blancs : int -> string = <fun>
# let disque taille =
      let moitié_droite = make_string taille '>'
      and moitié_gauche = make_string taille '<'
      in moitié_gauche ^ "|" ^ moitié_droite;;
disque : int -> string = <fun>
```

Ces deux fonctions construisent respectivement la représentation sous forme d'une chaîne d'une ligne vide de longueur `n` et d'un disque de largeur `taille`. Par exemple, le disque de largeur 3 est représenté par la chaîne `"<<<|>>>"` : la chaîne `"|"` représente un morceau de tige et `"<<<"` et `">>>"` les parties gauche et droite du disque.

La fonction suivante construit un disque, posé sur une tige, c'est-à-dire entouré d'un certain nombre d'espaces, de façon à ce que le disque occupe la même largeur que la tige sur laquelle il est posé :

```
# let disque_numéro n taille_grand_disque =
      let partie_blanche = blancs (taille_grand_disque + 1 - n) in
      partie_blanche ^ (disque n) ^ partie_blanche;;
disque_numéro : int -> int -> string = <fun>
```

La dernière fonction dessine la base d'une tige sous la forme `" ___|___ "`.

```
# let base_de_tige taille_grand_disque =
      let moitié = make_string taille_grand_disque '_' in
      " " ^ moitié ^ "|" ^ moitié ^ " ";;
base_de_tige : int -> string = <fun>
```

Un disque est représenté par un simple numéro : sa largeur. Une tige est représentée par un couple (entier, liste d'entiers). La première composante est le nombre de cases libres en haut de la tige, la seconde la liste des disques posés sur la tige. La fonction `tige` suivante construit la liste des chaînes de caractères représentant les disques posés sur une tige.

```
# let rec tige taille_grand_disque = function
    | (0, []) -> []
    | (0, tête :: reste) ->
        disque_numéro tête taille_grand_disque ::
```

```
        tige taille_grand_disque (0, reste)
    | (décalage, liste) ->
        disque_numéro 0 taille_grand_disque ::
        tige taille_grand_disque (décalage-1, liste);;
tige : int -> int * int list -> string list = <fun>
```

Par exemple, voici ce qu'on obtient pour la tige `(1, [2; 3; 5])`, c'est-à-dire une case libre, puis trois disques de largeur 2, 3 et 5 :

```
# let imprime ligne = print_string ligne; print_newline ();;
imprime : string -> unit = <fun>

# do_list imprime (tige 5 (1, [2; 3; 5]));;
      |
    <<|>>
   <<<|>>>
 <<<<<|>>>>>
- : unit = ()
```

Par la suite, nous allons avoir besoin d'afficher non pas une tige, mais trois tiges côte à côte. La fonction `recolle` crée la liste des lignes à afficher à partir des trois listes de lignes correspondant aux trois tiges.

```
# let rec recolle l1 l2 l3 =
     match l1, l2, l3 with
     | [], [], [] -> []
     | t1 :: r1, t2 :: r2, t3 :: r3 ->
        (t1 ^ t2 ^ t3) :: recolle r1 r2 r3
     | _ -> failwith "recolle";;
recolle : string list -> string list -> string list -> string list = <fun>
```

L'affichage d'une configuration consiste simplement à imprimer les lignes qui représentent les disques, puis à imprimer les trois bases des tiges.

```
# let imprime_jeu nombre_de_disques départ milieu arrivée =
      let dessin =
          recolle (tige nombre_de_disques départ)
                  (tige nombre_de_disques milieu)
                  (tige nombre_de_disques arrivée) in
      do_list imprime dessin;
      let b = base_de_tige nombre_de_disques in imprime (b ^ b ^ b);;
imprime_jeu :
 int -> int * int list -> int * int list -> int * int list -> unit = <fun>
```

Nous implémentons maintenant les déplacements de disques d'une tige à l'autre. Voyons tout d'abord la fonction qui ajoute un disque au sommet d'une tige. Par exemple, dans un jeu à quatre disques, ajouter le troisième disque à une tige qui ne comprend que le disque numéro 4 correspond à l'évaluation de `ajoute_disque 3 (2, [4])`, qui retourne `(1, [3; 4])`.

```
# let ajoute_disque disque (décalage, disques as tige) =
      (décalage - 1, disque :: disques);;
ajoute_disque : 'a -> int * 'a list -> int * 'a list = <fun>
```

On définit la fonction `sommet` pour consulter le disque qui se trouve au sommet d'une tige et la fonction `enlève_sommet` pour ôter le sommet d'une tige (plus exactement, pour renvoyer la tige privée de son sommet).

```
# let sommet = function
    | (décalage, sommet :: reste) -> sommet
    | (décalage, []) -> failwith "sommet: tige vide";;
sommet : 'a * 'b list -> 'b = <fun>
# let enlève_sommet = function
    | (décalage, sommet :: reste) -> (décalage + 1, reste)
    | (décalage, []) -> failwith "enlève_sommet: tige vide";;
enlève_sommet : int * 'a list -> int * 'a list = <fun>
```

Nous pouvons maintenant simuler un mouvement en déplaçant un disque d'une tige à l'autre. La procédure prend en argument des références sur les tiges concernées et les modifie physiquement au passage.

```
# let déplace (nom_départ, tige_départ) (nom_arrivée, tige_arrivée) =
      imprime("Je déplace un disque de " ^
              nom_départ ^ " à " ^ nom_arrivée);
      let disque_déplacé = sommet !tige_départ in
      tige_départ := enlève_sommet !tige_départ;
      tige_arrivée := ajoute_disque disque_déplacé !tige_arrivée;;
déplace :
 string * (int * 'a list) ref -> string * (int * 'a list) ref -> unit =
 <fun>
```

La modélisation de l'état initial du jeu nécessite la définition d'une tige vide et d'une tige pleine, en fonction du nombre de disques utilisés.

```
# let tige_vide nombre_de_disques = (nombre_de_disques, []);;
tige_vide : 'a -> 'a * 'b list = <fun>
# let tige_pleine nombre_de_disques =
      let rec liste_des_disques i =
          if i <= nombre_de_disques
          then i :: liste_des_disques (i + 1)
          else [] in
      (0, liste_des_disques 1);;
tige_pleine : int -> int * int list = <fun>
```

Nous définissons maintenant une procédure de résolution complète incluant les fonctions d'impression. Nous maintenons l'état des trois tiges à l'aide de trois références, `gauche`, `milieu` et `droite`. Remarquez que la procédure récursive locale `hanoi` prend ces trois références en argument et qu'elles sont modifiées physiquement par la procédure `déplace`.

```
# let jeu nombre_de_disques =
      let gauche = ref (tige_pleine nombre_de_disques)
      and milieu = ref (tige_vide nombre_de_disques)
      and droite = ref (tige_vide nombre_de_disques) in
      let rec hanoi hauteur départ intermédiaire destination =
          if hauteur > 0 then
           begin
             hanoi (hauteur - 1) départ destination intermédiaire;
             déplace départ destination;
             imprime_jeu nombre_de_disques !gauche !milieu !droite;
             hanoi (hauteur - 1) intermédiaire départ destination
           end in
```

```
# jeu 3;;
J'appelle les tiges A, B et C.
Position de départ:
   <|>       |        |
  <<|>>      |        |
 <<<|>>>     |        |
 ___|___  ___|___  ___|___
Je déplace un disque de A à C
    |        |        |
  <<|>>      |        |
 <<<|>>>     |       <|>
 ___|___  ___|___  ___|___
Je déplace un disque de A à B
    |        |        |
    |        |        |
 <<<|>>>   <<|>>     <|>
 ___|___  ___|___  ___|___
Je déplace un disque de C à B
    |        |        |
    |       <|>       |
 <<<|>>>   <<|>>      |
 ___|___  ___|___  ___|___
Je déplace un disque de A à C
    |        |        |
    |        |       <|>
    |        |      <<|>>
    |        |     <<<|>>>
 ___|___  ___|___  ___|___
Je déplace un disque de B à A
    |        |        |
    |        |        |
   <|>      <<|>>    <<<|>>>
 ___|___  ___|___  ___|___
Je déplace un disque de B à C
    |        |        |
    |        |       <<|>>
   <|>       |      <<<|>>>
 ___|___  ___|___  ___|___
Je déplace un disque de A à C
    |        |        |
    |        |       <|>
    |        |      <<|>>
    |        |     <<<|>>>
 ___|___  ___|___  ___|___
- : unit = ()
```

Figure 5.1: Résolution des tours de Hanoi à 3 disques

```
      imprime "J'appelle les tiges A, B et C.";
      imprime "Position de départ:";
      imprime_jeu nombre_de_disques !gauche !milieu !droite;
      hanoi nombre_de_disques
            ("A", gauche) ("B", milieu) ("C", droite);;
jeu : int -> unit = <fun>
```

La figure 5.1 montre le déroulement de `jeu 3`.

5.9 Fonctionnelles complexes sur les listes

Nous allons maintenant passer en revue un certain nombre de fonctionnelles classiques sur les listes. Elles sont d'un emploi plus rare que celles que nous avons déjà vues. Nous les utiliserons dans les exemples les plus difficiles. Vous pourrez alors vous reporter aux explications qui suivent.

Notion d'abstraction de schémas de programmes

Vous vous souvenez sans doute que pour généraliser les fonctions de tri du chapitre 4 et du présent chapitre, il nous avait suffi de passer la comparaison en paramètre. Nous avions dit que nous avions « abstrait » l'ordre. Les fonctionnelles que nous allons décrire sont également basées sur la notion d'abstraction. Cependant, dans ce cas il ne s'agit pas d'abstraire une fonction qui intervient dans le programme pour le généraliser, mais

d'abstraire un *schéma de programme*, c'est-à-dire une méthode de calcul commune à plusieurs algorithmes. Pour dégager cette méthode de calcul, nous allons montrer plusieurs fonctions qui l'utilisent. Nous rappelons d'abord le code des fonctions `somme` et `produit`, définies ci-dessus ; puis nous écrivons la fonction `implose`, qui renvoie la concaténation de toutes les chaînes d'une liste et enfin la fonction `concatène_listes` qui concatène toutes les listes d'une liste de listes (ce dernier exemple utilise la fonction prédéfinie `@` qui concatène deux listes).

```
# let rec somme = function
    | [] -> 0
    | x :: l -> x + somme l;;
somme : int list -> int = <fun>
```

```
# let rec produit = function
    | [] -> 1
    | x :: l -> x * produit l;;
produit : int list -> int = <fun>
```

```
# let rec implose = function
    | [] -> ""
    | x :: l -> x ^ implose l;;
implose : string list -> string = <fun>
# implose  ["Bonjour"; "tout"; "le "; "monde!"];;
- : string = "Bonjourtoutle monde!"
# let rec concatène_listes = function
    | [] -> []
    | x :: l -> x @ concatène_listes l;;
concatène_listes : 'a list list -> 'a list = <fun>
# concatène_listes [[1]; [2; 3]; [4; 5; 6]];;
- : int list = [1; 2; 3; 4; 5; 6]
```

On constate que toutes ces fonctions utilisent le même schéma récursif : en cas de liste vide, on rend un certain élément de base ; en cas de liste non vide, on appelle une fonction à deux arguments, avec pour premier argument la tête de la liste et pour second argument un appel récursif sur la fin de la liste. Pour `somme` l'élément de base est `0` et l'opération est `+`, pour `produit` c'est `1` et `*`, pour `implose` c'est `""` et `^`, enfin pour `concatène_listes` c'est `[]` et `@`. Pour obtenir notre fonctionnelle générale qui implémente ce schéma de programme, il nous suffit donc d'abstraire la fonction et l'élément de base.

Une petite remarque syntaxique au préalable : toutes les opérations utilisées ici sont infixes. Pour écrire la fonctionnelle, nous utilisons un argument fonctionnel normal (donc préfixe). Il faut donc relire le code des exemples avec des opérations préfixes : par exemple pour `somme`, l'opération préfixe correspondant à `+` est la fonction prédéfinie `add_int`, c'est-à-dire `function x -> function y -> x + y`. Notre fonctionnelle s'écrit maintenant très facilement :

```
# let rec itérateur_sur_listes f b = function
    | [] -> b
    | x :: l -> f x (itérateur_sur_listes f b l);;
itérateur_sur_listes : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
```

Pour définir `somme` en utilisant l'itérateur, on écrira simplement :

```
# let somme l = itérateur_sur_listes add_int 0 l;;
somme : int list -> int = <fun>
```

```
# somme [1; 2; 3];;
- : int = 6
```

D'autres programmes ressortissent du même schéma. Considérons par exemple la fonction qui calcule la longueur d'une liste, qui est prédéfinie en Caml sous le nom de `list_length`. Elle s'écrit sans difficulté :

```
# let rec list_length = function
    | [] -> 0
    | x :: l -> 1 + list_length l;;
list_length : 'a list -> int = <fun>
```

C'est encore le même schéma : l'élément de base est 0, l'opération est l'addition de 1. On peut donc écrire

```
# let list_length l =
      itérateur_sur_listes (function x -> function y -> 1 + y) 0 l;;
list_length : 'a list -> int = <fun>
# list_length [0; 3; 5; 7];;
- : int = 4
```

On écrit de même la concaténation des listes, à l'aide d'une fonction auxiliaire `devant` qui recopie une liste devant une autre.

```
# let rec devant l2 = function
    | [] -> l2
    | x :: l -> x :: devant l2 l;;
devant : 'a list -> 'a list -> 'a list = <fun>
# devant [4; 5; 6] [1; 2; 3];;
- : int list = [1; 2; 3; 4; 5; 6]
# let concatène l1 l2 = devant l2 l1;;
concatène : 'a list -> 'a list -> 'a list = <fun>
```

La fonction `devant` suit également le même schéma, avec élément de base `l2` et opération « `::` ».

```
# let devant l2 =
      itérateur_sur_listes (function x -> function y -> x :: y) l2;;
devant : 'a list -> 'a list -> 'a list = <fun>
# devant [4; 5; 6] [1; 2; 3];;
- : int list = [1; 2; 3; 4; 5; 6]
```

Notion d'accumulation récursive

Un autre schéma récursif qui revient souvent est l'accumulation. Par exemple, pour écrire la fonction `somme`, il est naturel d'ajouter les éléments de la liste à un accumulateur, qu'on rendra en résultat quand on arrivera à la fin de la liste. Nous sommes donc amenés à écrire la fonction `somme_accu` suivante, qui accumule les éléments d'une liste dans son accumulateur argument `accu` :

```
# let rec somme_accu accu = function
    | [] -> accu
    | x :: l -> somme_accu (x + accu) l;;
somme_accu : int -> int list -> int = <fun>
```

Il est clair qu'on obtient la somme des éléments d'une liste en appelant `somme_accu` avec un accumulateur valant initialement 0.

```
# let somme l = somme_accu 0 l;;
somme : int list -> int = <fun>
# somme [1; 2; 3];;
- : int = 6
```

La fonctionnelle générale correspondant au code de `somme_accu` est simplement :

```
# let rec accumulateur_sur_listes f accu = function
    | [] -> accu
    | x :: l -> accumulateur_sur_listes f (f x accu) l;;
accumulateur_sur_listes : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
```

La fonction `somme_accu` s'obtient maintenant en utilisant `accumulateur_sur_listes` avec l'addition :

```
# let somme_accu l =
    accumulateur_sur_listes
      (function x -> function accu -> x + accu) l;;
somme_accu : int -> int list -> int = <fun>
# let somme l = somme_accu 0 l;;
somme : int list -> int = <fun>
# somme [1; 2; 3];;
- : int = 6
```

Les fonctionnelles prédéfinies en Caml sont analogues à celles que nous venons de dégager, avec cependant des différences minimes. La fonctionnelle `itérateur_sur_listes` correspond à `list_it` et `accumulateur_sur_listes` est équivalente à `it_list`. Nous décrivons maintenant ces fonctionnelles prédéfinies en leur donnant une interprétation graphique.

Accumuler avec les éléments d'une liste

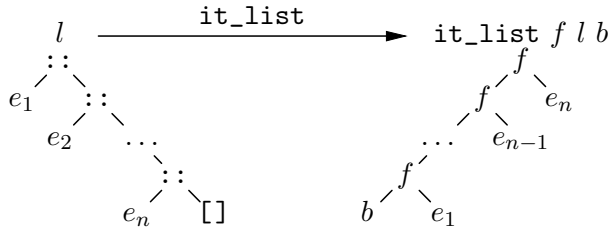
Étant donnés trois arguments f, b et l, la fonctionnelle `it_list` effectue de multiples compositions de la fonction à deux arguments f, en utilisant les éléments de la liste l comme seconds arguments de f. La valeur de base b est utilisée pour le premier argument du premier appel à f, puis le résultat de chaque appel à f est passé en premier argument de l'appel suivant. `it_list` est caractérisée par :

$$\texttt{it_list}\ f\ b\ [e_1;\ e_2;\ \ldots\ ;\ e_n] = (f\ (\ldots\ (f\ (f\ b\ e_1)\ e_2)\ \ldots)\ e_n).$$

Le code de `it_list` est :

```
# let rec it_list f b = function
    | [] -> b
    | x :: l -> it_list f (f b x) l;;
it_list : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

Intuitivement, si l'on peut dire, `it_list` fait de la « réécriture de liste » : elle remplace tous les `;` de la liste par f (considérée comme un opérateur binaire infixe) et ajoute la valeur initiale b au premier appel à f. Graphiquement, l'effet calculatoire de `it_list` se représente par la transformation :



Par exemple, supposons que f soit l'addition `+`. Nous obtenons

$$\texttt{it_list}\ f\ b\ [e_1;\ e_2;\ \ldots\ ;\ e_n] = b\ +\ e_1\ +\ e_2\ +\ \cdots\ +\ e_n.$$

Maintenant nous pouvons écrire la fonction `somme` en utilisant `it_list` :

```
# let ajoute x y = x + y;;
ajoute : int -> int -> int = <fun>
# let somme l = it_list ajoute 0 l;;
somme : int list -> int = <fun>
# somme [1; 2; 3];;
- : int = 6
```

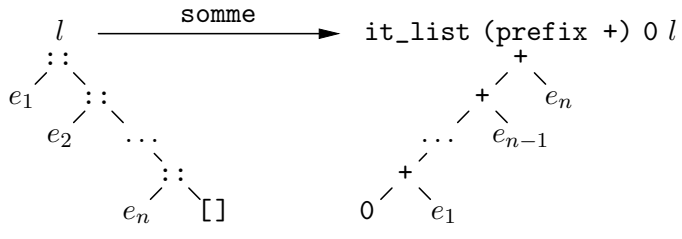
Allons encore un peu plus loin : il est inutile de définir la fonction `ajoute` qui est équivalente à l'opérateur infixe `+`. En effet, en Caml on fait référence à la forme préfixe d'un opérateur infixe (comme `+`) en le faisant simplement précéder du mot-clé `prefix` :

```
# (prefix +);;
- : int -> int -> int = <fun>
# (prefix +) 1 2;;
- : int = 3
```

Cette facilité conduit à des définitions de `somme` et `produit` en une ligne (après une étape de η-contraction) :

```
# let somme = it_list (prefix +) 0
  and produit = it_list (prefix *) 1;;
somme : int list -> int = <fun>
produit : int list -> int = <fun>
```

Graphiquement, cela s'exprime par les deux réécritures suivantes :



Est-ce plus clair que notre première version de `somme` définie par filtrage sur les listes ? Pas pour les auteurs de ce livre en tout cas : nous pensons que cette écriture semblera souvent obscure à d'autres programmeurs, y compris vous-même trois mois après avoir écrit ce code ! Si vous voulez être lisible, évitez ce style de « programmation sans variables », particulièrement si vous êtes capable d'écrire du code qui utilise encore plus de fonctionnelles que celui de `somme` : ce code peut devenir trop « élégant » pour être maintenu ...

Accumuler encore

Étant donnés trois arguments f, b et l, la fonctionnelle `list_it` effectue de multiples compositions de la fonction à deux arguments f, en utilisant les éléments de la liste l comme premiers arguments de f ; l'élément de base b est utilisé pour le second argument du premier appel à f, puis le résultat de chaque appel à f est passé en second argument de l'appel suivant. `list_it` est caractérisée par :

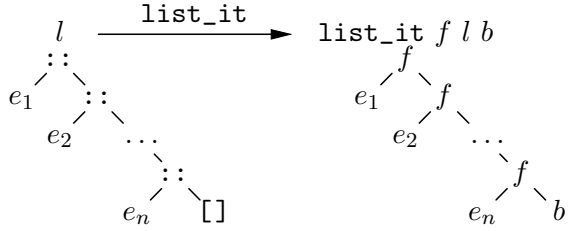
$$\texttt{list_it}\ f\ [e_1;\ e_2;\ \ \ldots\ ;\ e_n]\ b = f\ e_1\ (f\ e_2(\ \ldots\ (f\ e_n\ b)\ \ldots\ )).$$

En termes encore plus savants et en utilisant l'opérateur mathématique de composition des fonctions $\circ$, on écrirait :

$$\texttt{list_it}\ f\ [e_1;\ e_2;\ \ \ldots\ ;\ e_n]\ b\ =\ ((f\ e_1) \circ (f\ e_2) \circ \cdots \circ (f\ e_n))\ (b)$$

```
# let rec list_it f l b =
      match l with
      | [] -> b
      | x :: reste -> f x (list_it f reste b);;
list_it : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

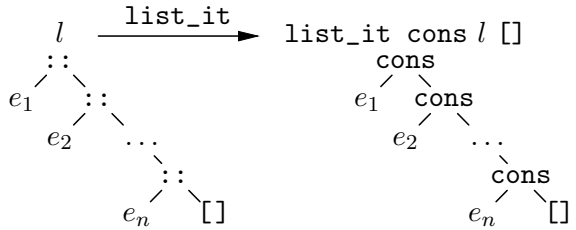
Intuitivement, `list_it` fait aussi de la « réécriture de liste » : elle remplace toutes les occurrences de « `::` » par f et le « `[]` » final par la valeur initiale b.



Par exemple, la fonction qui copie une liste est facile à définir : elle doit remplacer tous les `::` de la liste par d'autres `::` et le `[]` final par `[]`. La fonction f qui fait ce travail de ne rien modifier et donc de remplacer `::` par `::`, est simplement la fonction `function x -> function l -> x :: l`. Nous l'appelons `cons` et la définissons ainsi :

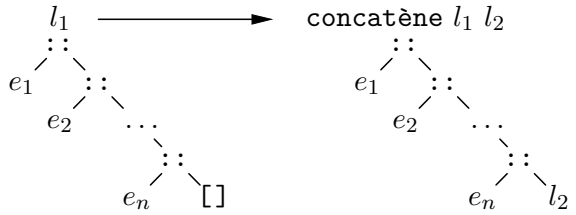
```
# let cons x l = x :: l;;
cons : 'a -> 'a list -> 'a list = <fun>
```

Avec `list_it`, nous écrivons facilement la fonction de copie :



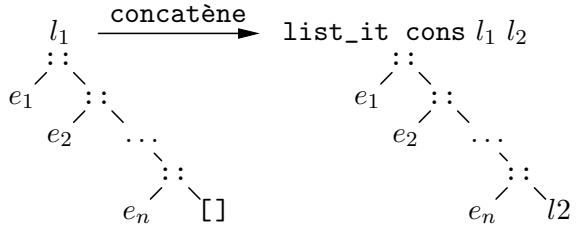

```
# let copie_liste l = list_it cons l [];;
copie_liste : 'a list -> 'a list = <fun>
# copie_liste [1;2;3];;
- : int list = [1; 2; 3]
```

Cette fonction n'est pas vraiment utile, sauf pour copier une liste devant une autre. En effet, si nous voulons maintenant concaténer deux listes l_1 et l_2 (mettre les deux listes bout à bout), il suffit de mettre l_2 à la fin de l_1, donc de remplacer le `[]` final de l_1 par toute la liste l_2.



Il suffit donc d'appeler `list_it` sur l_1 avec la fonction `cons` et l'élément final l_2.

```
# let concatène l1 l2 = list_it cons l1 l2;;
concatène : 'a list -> 'a list -> 'a list = <fun>
# concatène [1; 2; 3] [4; 5; 6];;
- : int list = [1; 2; 3; 4; 5; 6]
```



Définitions récursives locales

Nous revenons sur le code de la fonction `map` pour faire la remarque suivante : étant donnée une fonction f, `map` se contente de boucler sur une liste, en appliquant f. C'est pourquoi il nous suffit de définir une fonction récursive *à l'intérieur* de `map`, qui saura appliquer f sur les éléments d'une liste quelconque :

```
# let map f =
      let rec map_fonction_f = function
        | [] -> []
        | x :: l -> f x :: map_fonction_f l in
      (function liste -> map_fonction_f liste);;
map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Ainsi, `map` n'est plus récursive, mais comporte une définition locale de fonction récursive. On simplifie encore ce code en utilisant la règle η, qui stipule que

```
(function liste -> map_fonction_f liste)
```

est équivalent à la forme plus simple `map_fonction_f`. On obtient alors le code suivant :

```
# let map f =
      let rec map_fonction_f = function
        | [] -> []
        | x :: l -> f x :: map_fonction_f l in
      map_fonction_f;;
map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Cette vision de `map` correspond à celle d'une fonctionnelle qui, appliquée à une fonction f, retourne la fonction qui itère f sur une liste. Avec cette vision nous pourrions écrire :

```
# let successeur x = x + 1;;
successeur : int -> int = <fun>
# let liste_des_successeurs = map successeur;;
liste_des_successeurs : int list -> int list = <fun>
# liste_des_successeurs [0; 1; 2];;
- : int list = [1; 2; 3]
```

Ce style de programmation n'est pas à encourager : la définition naïve de `map` nous paraît plus claire. D'autre part, la définition de `map` avec une fonction locale ne se justifie pas non plus par un gain majeur en efficacité : suivant les compilateurs, elle sera un peu plus ou un peu moins efficace que la définition naïve.

En revanche, dans le cas où une fonction récursive prend beaucoup de paramètres en argument et où ces paramètres sont inchangés dans les appels récursifs, on peut admettre de définir localement une fonction récursive qui s'appelle avec les seuls paramètres modifiés. En ce cas, on remplacerait la définition d'une fonction `f` à plusieurs paramètres `x1`, `x2`, ... , `xn`, dont les appels récursifs ne font varier que `xn`, par une définition comportant une fonction locale ayant `xn` pour seul paramètre. Ainsi

```
let rec f x1 x2 x3 ... xn =
    ... f x1 x2 x3...(xn + 1) ... f x1 x2 x3...(xn - 1) ...;;
```

deviendrait

```
let f x1 x2 x3 ... =
    let rec f_locale xn =
    ... f_locale (xn + 1) ... f_locale (xn - 1) ... in
    f_locale;;
```

Ce style ne se justifie que pour des raisons de concision ; il ne doit pas être érigé en système.

5.10 Efficacité des fonctions sur les listes : étude de cas

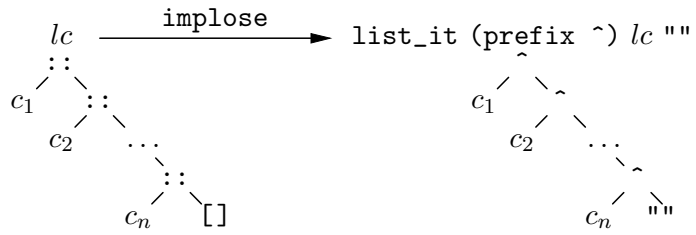
Dans cette section, nous analysons la complexité de deux fonctions, `implose` et `rev`. L'étude de complexité nous amène à écrire des versions plus efficaces, en style impératif pour `implose` et en style purement fonctionnel pour `rev`. L'amélioration obtenue est basée dans les deux cas sur l'emploi d'un accumulateur qui maintient les résultats

intermédiaires de la fonction. L'optimisation ne dépend donc pas obligatoirement du style de programmation utilisé, puisqu'une même idée sert dans les deux styles et pour des programmes différents.

La fonction `implose`, version fonctionnelle

La fonction `implose` concatène toutes les chaînes d'une liste de chaînes. Nous en avons déjà écrit la version la plus naïve dans la section 5.9. Raisonnons graphiquement : pour obtenir la concaténation de toutes les chaînes d'une liste il faut « réécrire » la liste en remplaçant les « `::` » par des « `^` » et le symbole `[]` final par la chaîne vide. Algébriquement, si l'on note lc la liste argument, $[c_1;\ c_2;\ \ldots\ ;\ c_n]$, on a

$$\begin{aligned} \texttt{implose}\,[c_1;\ c_2;\ \ldots\ ;\ c_n] &= c_1\,\texttt{\^{}}\,c_2\,\texttt{\^{}}\,\ldots\,\texttt{\^{}}\,c_n\,\texttt{\^{}}\,\texttt{""} \\ &= \texttt{list_it (prefix \^{})}\,lc\,\texttt{""} \end{aligned}$$



De la même façon, mais en utilisant `it_list` au lieu de `list_it` :

$$\begin{aligned} \texttt{implose}\,[c_1;\ c_2;\ \ldots\ ;\ c_n] &= \texttt{""}\,\texttt{\^{}}\,c_1\,\texttt{\^{}}\,\ldots\,\texttt{\^{}}\,c_n \\ &= \texttt{it_list (prefix \^{}) ""}\,lc \end{aligned}$$

On obtient donc :

```
# let implose lc = list_it (prefix ^) lc "";;
implose : string list -> string = <fun>
# let implose2 lc = it_list (prefix ^) "" lc;;
implose2 : string list -> string = <fun>
```

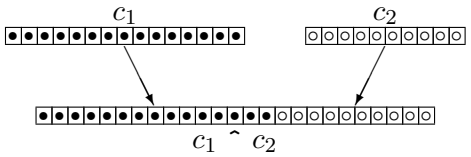
`implose2` se simplifie par la règle η, pour donner un code extrêmement similaire à celui de `somme` (opération binaire `prefix ^` au lieu de `prefix +` et élément neutre `""` au lieu de `0`) :

```
# let implose2 = it_list (prefix ^) "";;
implose2 : string list -> string = <fun>
```

Cette écriture est extrêmement compacte ; on peut même la considérer comme élégante. Cependant l'emploi des itérateurs, en produisant un code compact, a tendance à cacher la complexité des algorithmes. Nous allons voir que notre fonction `implose` a une complexité élevée (quadratique en le nombre de chaînes concaténées), à cause de la création de nombreuses chaînes intermédiaires.

L'opérateur de concaténation de chaînes

Pour calculer la complexité de la fonction `implose`, il nous faut réfléchir sur le fonctionnement de l'opérateur `^` de concaténation de chaînes. Étant données deux chaînes de caractères c_1 et c_2 en arguments, `^` alloue une nouvelle chaîne de caractères pour y loger la concaténation des deux chaînes, puis y recopie c_1 et c_2 correctement décalées.



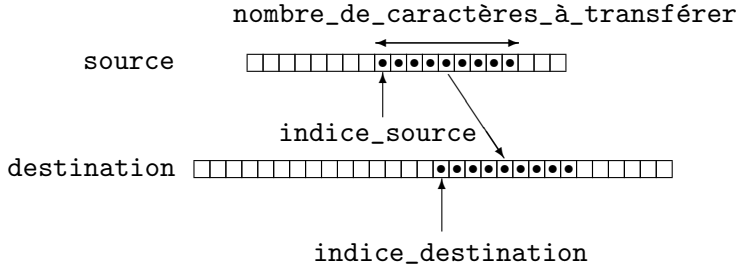
Nous écrivons l'opérateur `^` sous la forme d'une fonction préfixe `concat` :

```
# let concat c1 c2 =
    let résultat =
      create_string (string_length c1 + string_length c2) in
    blit_string c1 0 résultat 0 (string_length c1);
    blit_string c2 0 résultat (string_length c1) (string_length c2);
    résultat;;
concat : string -> string -> string = <fun>
```

On rappelle que l'appel de la procédure

```
blit_string source indice_source destination indice_destination
            nombre_de_caractères_à_transférer
```

transfère `nombre_de_caractères_à_transférer` dans la chaîne `destination` à partir de l'indice `indice_destination`. Ces caractères sont ceux de la chaîne `source` à partir de l'indice `indice_source`.



Pour calculer la complexité de la fonction `implose`, nous considérerons que le coût d'une concaténation est indépendant de la longueur des deux chaînes concaténées, ou encore que toutes les chaînes sont d'égale longueur et même de longueur 1 pour simplifier encore. Nous supposons que la liste argument lc comprend n chaînes c_1, c_2, $\ldots c_n$. À chaque chaîne c_i de la liste de chaînes argument, on recopie le précédent résultat dans une nouvelle chaîne f, puis on recopie la chaîne c_i dans f. On alloue donc autant de chaînes que de résultats intermédiaires, soit n. De plus, c_1 est recopiée une fois, c_2 deux fois, et c_n est recopiée n fois. Ce qui donne donc $1 + 2 + 3 + \cdots + n$ caractères copiés. Or, un calcul facile (il suffit d'ajouter les termes de la somme en les groupant astucieusement, premier avec dernier, deuxième avec avant-dernier, etc.) montre que

$$1 + 2 + 3 + \cdots + n \;=\; \frac{n \times (n+1)}{2}$$

Quand n est grand, $(n \times (n+1))/2$ est proche de $n^2/2$. Notre fonction a donc une complexité de l'ordre de n^2 : elle est quadratique. Nous avons déjà vu que c'est une complexité importante. Nous allons tenter de l'améliorer.

La fonction `implose`, version impérative

L'idée, très simple, est de calculer d'abord la longueur de la chaîne nécessaire au résultat final, de l'allouer, puis d'y copier toutes les chaînes de la liste.

```
# let implose chaînes =
    let longueur = ref 0 in
    do_list
      (function ch ->
        longueur := string_length ch + !longueur)
      chaînes;
    let résultat = create_string !longueur
    and position = ref 0 in
    do_list
      (function ch ->
        blit_string ch 0 résultat !position (string_length ch);
        position := !position + string_length ch)
      chaînes;
    résultat;;
implose : string list -> string = <fun>
```

La complexité de cette version est alors d'une seule allocation de chaîne et de n copies de caractères : cette version est donc linéaire. La différence d'efficacité est énorme : pour 10^4 chaînes de caractères, la première version nécessite $10^8/2$ copies, soit 5000 fois plus que l'algorithme linéaire. En d'autres termes, si la version linéaire demande 1 seconde pour réaliser la concaténation des 10^4 chaînes, alors la version quadratique nécessite plus de 1 *heure* de calcul !

Nous avons ainsi optimisé la fonction `implose` en passant dans le monde impératif. Cette démarche n'est pas obligatoire ; l'optimisation consiste aussi à remplacer un algorithme du monde fonctionnel par un autre plus efficace mais sans quitter le monde fonctionnel. C'est ce que nous allons voir maintenant.

Retournement d'une liste

La fonction `rev` est un exemple emblématique de fonction simple qui a pourtant un comportement catastrophique si l'on n'a pas le souci de réfléchir à la complexité de ses programmes. La fonction `rev` renvoie sa liste argument à l'envers. Ici, l'analyse de complexité nous suggère un programme fonctionnel bien meilleur.

On écrit une version très naïve de `rev` en se basant sur le raisonnement suivant :

- Si la liste est vide, son « envers » est aussi vide.
- Si la liste n'est pas vide, il suffit d'ajouter son premier élément à la fin du reste de la liste à l'envers.

Cela se traduit immédiatement par la définition :

```
# let rec rev = function
    | [] -> []
    | x :: l -> concatène (rev l) [x];;
rev : 'a list -> 'a list = <fun>
# rev [1; 2; 3];;
- : int list = [3; 2; 1]
```

Cette version est encore plus jolie avec la version infixe prédéfinie de `concatène`, l'opérateur `@`.

```
# let rec rev = function
    | [] -> []
    | x :: l -> rev l @ [x];;
rev : 'a list -> 'a list = <fun>
```

Cet algorithme est simple, le programme extrêmement concis, mais malheureusement inefficace, car on ne cesse de recopier des listes en utilisant la fonction `concatène`. Suivons le déroulement de l'évaluation du renversement de la liste `[1; 2; 3]` :

```
rev [1 ; 2 ; 3]
  →  concatène (rev [2 ; 3]) [1]
  →  concatène (concatène (rev [3]) [2]) [1]
  →  concatène (concatène (concatène (rev []) [3]) [2]) [1]
  →  concatène (concatène (concatène [] [3]) [2]) [1]
  →  concatène (concatène [3] [2]) [1]
  →  concatène (3 :: [2]) [1]
  →  concatène [3 ; 2] [1]
  →  3 :: 2 :: [1]
  →  [3 ; 2 ; 1]
```

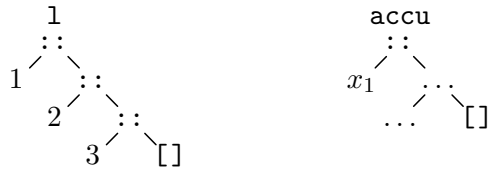
Il apparaît que le premier élément de la liste argument, 1, a été « consé » une fois (pour fabriquer la liste `[1]`). Le second, 2, a été « consé » deux fois (une fois pour fabriquer la liste `[2]` et une fois pour obtenir la liste intermédiaire `[2; 1]`). Le troisième, 3, a été « consé » trois fois. On montre facilement que si la liste l a n éléments, cet algorithme crée une cellule de liste pour le premier élément, deux pour le second, ..., et finalement n cellules pour le dernier élément. Le nombre total de cellules crées est donc encore la somme $1 + 2 + 3 + \cdots + n$, qui vaut $n \times (n + 1)/2$: l'algorithme est quadratique. Le phénomène de copie récursive des résultats partiels conduisant à un algorithme en n^2 est le même que pour la fonction `implose`.

Pour améliorer cet algorithme, il faut utiliser une méthode très générale : pour construire directement le résultat, on ajoute un argument supplémentaire à la fonction. Cet argument joue le rôle d'un accumulateur, car on y mémorise les résultats partiels de la fonction lors des appels récursifs. Cela conduit à écrire une fonction elle aussi plus générale, dans la mesure où il est possible maintenant de l'appeler avec un accumulateur non vide au départ. Prenant un peu de recul, on définit donc une fonction auxiliaire `concatène_à_l'envers`, qui étant donnés une liste et un accumulateur, recopie la liste en tête de l'accumulateur :

```
# let rec concatène_à_l'envers accu = function
    | [] -> accu
    | x :: l -> concatène_à_l'envers (x :: accu) l;;
concatène_à_l'envers : 'a list -> 'a list -> 'a list = <fun>
```

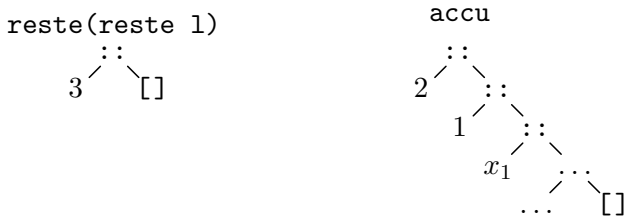
```
# concatène_à_l'envers [0] [1; 2; 3];;
- : int list = [3; 2; 1; 0]
```

Illustrons graphiquement le comportement de cette fonction. On part d'une liste `l` et d'un accumulateur `accu` qui contient éventuellement déjà des éléments :



Puis on recopie la tête de la liste `l` dans l'accumulateur, obtenant :

À l'étape suivante on aura :



Il est clair maintenant que l'accumulateur engrange les éléments de la liste `l` à l'envers. La fonction `rev` s'en déduit simplement, en appelant `concatène_à_l'envers` avec un accumulateur vide :

```
# let rev l = concatène_à_l'envers [] l;;
rev : 'a list -> 'a list = <fun>
# rev [1; 2; 3];;
- : int list = [3; 2; 1]
```

Notre algorithme est maintenant linéaire. Cette méthode d'accumulation des résultats intermédiaires dans un argument supplémentaire de la fonction est souvent une bonne piste à suivre pour optimiser une fonction, quand cette fonction pose des problèmes d'efficacité. Encore faut-il prouver que la fonction sujette à optimisation est réellement le goulet d'étranglement du programme, puis démontrer par une étude de complexité que l'optimisation va vraiment améliorer les performances. C'est évidemment très difficile. En règle générale, on se contentera d'écrire des programmes corrects et lisibles.

5.11 Listes et récurrence

Nous montrons dans cette section comment prouver des propriétés sur les listes. Bien qu'un peu théorique, ce n'est pas très complexe, puisqu'il s'agit d'une extension simple du principe de récurrence.

Lorsque nous avons écrit des fonctions récursives sur les entiers, nous apportions la preuve de leurs propriétés en utilisant le principe de récurrence. Mais ce principe ne s'applique plus dans le cas des listes, puisqu'il concerne uniquement les propriétés définies sur les nombres entiers. Comment prouver des propriétés des listes ? Il suffit de se ramener au cas des entiers en raisonnant sur le nombre entier qui mesure la longueur de la liste. Le principe de récurrence nous permet alors de démontrer qu'une propriété est vraie pour des listes de n'importe quelle longueur, donc pour toutes les listes. Une application directe du principe de récurrence nous permet ainsi d'établir que :

> Si une propriété P est vraie pour une liste de longueur 0, et si dès qu'elle est vraie pour une liste de longueur n elle est vraie pour une liste de longueur $n+1$, alors P est vraie pour des listes de n'importe quelle longueur.

Si l'on remarque qu'il n'existe qu'une seule liste de longueur 0, la liste vide, et qu'une liste de longueur $n+1$ s'obtient forcément en rajoutant un élément à une liste de longueur n, on obtient maintenant l'énoncé :

> Si une propriété P est vraie pour `[]` et si dès que P est vraie pour l alors P est vraie pour $x::l$, alors P est vraie pour toutes les listes.

C'est ce qu'on appelle le principe de récurrence structurelle sur les listes. Remarquons que ces deux cas, `[]` et $x::l$, sont justement les deux cas du filtrage d'une fonction récursive sur les listes. Cela justifie nos raisonnements informels précédents, quand nous disions « un petit appel récursif et le tour est joué », ou bien que nous appelions récursivement une fonction pas encore écrite en pensant « qu'elle saurait bien faire toute seule ». Cela justifie aussi le qualificatif « structurelle » de ce principe de récurrence, puisqu'on raisonne en fait sur la structure des listes.

Prouvons par exemple que la fonction `insère`, qui nous a servi de fonction auxiliaire pour le tri par insertion, insère correctement un élément dans une liste. Nous considérons donc la propriété $P(l)$ suivante : sous l'hypothèse que l est une liste bien triée, `insère` *élément* l est une liste, elle aussi bien triée, qui comprend *élément* et tous les éléments de la liste l.

1. P est vraie pour `[]`. En effet, `insère` *élément* `[]` vaut `[`*élément*`]`, qui est forcément bien triée, comprend *élément* et tous les éléments de la liste vide.
2. Supposons $P(l)$ vraie. Alors $P(x::l)$ est vraie aussi. En effet, d'après la définition de la fonction `insère`, si *élément* $\leq x$ alors `insère` *élément* $(x::l)$ vaut *élément*$::x::l$, qui contient *élément* et tous les éléments de la liste argument $x::l$, et ce résultat est bien trié puisque $x::l$ est bien triée par hypothèse et que *élément* $\leq x$. Dans le cas où *élément* $> x$, alors `insère` *élément* $(x::l)$ vaut $x::$(`insère` *élément* l). Cette liste est bien triée car, d'après l'hypothèse de récurrence, $P(l)$ est vraie, donc (`insère` *élément* l) est bien triée ; mais x est le plus petit élément de $x::$(`insère` *élément* l), puisque c'était déjà le plus petit élément de $(x::l)$ et qu'il est plus petit que *élément*. De plus la liste $x::$(`insère` *élément* l) contient *élément* et tous les éléments de $(x::l)$ car elle contient évidemment x et par hypothèse de récurrence (`insère` *élément* l) contient tous les éléments de l et l'élément à insérer *élément*.

En conclusion, notre fonction `insère` fonctionne : P est vraie pour toute liste. Donc, si l est une liste bien triée, `insère` *élément* l est bien triée et comprend *élément* en plus

de tous les éléments de la liste l.

Les propriétés des fonctions définies sur les listes se démontreront toujours de façon analogue, en suivant le filtrage utilisé par la fonction pour diriger la preuve par induction structurelle (induction signifie démonstration par récurrence).

5.12 À la recherche de l'itérateur unique

Nous avons réussi à exprimer la fonction `implose` en fonction de `it_list` et de `list_it`. La question se pose donc du choix de l'itérateur le plus efficace et de l'utilité d'en avoir deux.

Récursivité terminale

En ce qui concerne l'efficacité, `it_list` est légèrement plus efficace que `list_it`, car il est *récursif terminal*, ce qui signifie qu'il peut s'implémenter en machine par un simple saut : il ne laisse pas de calculs en suspens. Par exemple, la fonctionnelle `do_list` est récursive terminale :

```
# let rec do_list f = function
    | [] -> ()
    | x :: l -> f x; do_list f l;;
do_list : ('a -> 'b) -> 'a list -> unit = <fun>
```

En effet, après avoir exécuté `f x` on rappelle directement `do_list` en oubliant le calcul précédent. En revanche, `map` n'est pas récursive terminale :

```
# let rec map f = function
    | [] -> []
    | x :: l -> f x :: map f l;;
map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Lors de l'appel récursif `map f l`, il faut sauvegarder quelque part la valeur de `f x` pour l'ajouter en tête du résultat de `map f l`. Généralement, ce résultat intermédiaire est sauvegardé dans un tableau en attendant le retour de l'appel récursif. Ce tableau est appelé *pile d'exécution* du programme. Comme toutes les ressources mémoire, la pile est de taille finie et une fonction qui travaille en espace de pile constant comme `do_list` est préférable à une fonction qui consomme de la pile comme `map`. Lorsqu'on a le choix entre une fonction récursive terminale et une autre qui ne l'est pas, on préfère généralement celle qui est récursive terminale, pourvu qu'elle reste simple : rendre une récursion terminale ne justifie généralement pas qu'on complique le programme.

Itérateurs et effets

L'itérateur `list_it` est très puissant : en ce qui concerne les calculs proprement dits, il n'est pas nécessaire d'en avoir d'autre. Par exemple, `map` s'écrit très facilement avec `list_it`.

```
# let map f l =
      list_it (function x -> function res -> f x :: res) l [];;
map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
# map successeur [1; 2; 3];;
- : int list = [2; 3; 4]
```

Dans le même esprit il serait tentant d'écrire `do_list` en fonction de `list_it`.

```
# let do_list f l =
      list_it (function x -> function y -> f x; y) l ();;
do_list : ('a -> 'b) -> 'a list -> unit = <fun>

# do_list print_int [1; 2; 3];;
321- : unit = ()
```

La fonction est bien appliquée sur tous les éléments de la liste, mais à l'envers. Les effets se produisent donc dans l'ordre inverse de la vraie fonction `do_list`. Cependant, il suffit d'exprimer `do_list` en fonction de `it_list` pour que tout rentre dans l'ordre.

```
# let do_list f l =
      it_list (function y -> function x -> f x; y) () l;;
do_list : ('a -> 'b) -> 'a list -> unit = <fun>

# do_list print_int [1; 2; 3];;
123- : unit = ()
```

Tentons alors d'exprimer également `map` en fonction de `it_list`.

```
# let map f l =
      it_list (function res -> function x -> f x :: res) [] l;;
map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Malheureusement, la liste résultat n'est pas dans le bon ordre.

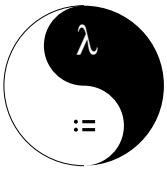
```
# map successeur [1; 2; 3];;
- : int list = [4; 3; 2]

# map (function x -> print_int x; successeur x) [1; 2; 3];;
123- : int list = [4; 3; 2]
```

En effet, `it_list` accumule les résultats dans la liste `res` en les ajoutant en tête de liste. Comme dans le cas de `rev`, on obtient ainsi la liste des résultats à l'envers. D'ailleurs, si l'on ne fait qu'accumuler les éléments sur la liste des résultats précédents, on obtient effectivement une autre version de `rev` :

```
# let rev l = it_list (function res -> function x -> x :: res) [] l;;
rev : 'a list -> 'a list = <fun>

# rev [1; 2; 3];;
- : int list = [3; 2; 1]
```

Cette version peu commune de `rev` est également linéaire et récursive terminale.

En conclusion, on constate que `map` et `do_list` sont des versions spécialisées d'itérateurs plus généraux. Ces fonctions gardent cependant leur intérêt, car elles sont simples à employer et à comprendre. Du point de vue purement calculatoire, `it_list` et `list_it` sont un tant soit peu redondants, mais ils se distinguent lorsque les fonctions qu'ont leur applique produisent des effets. On constate une fois de plus que les effets compliquent les choses, car ils permettent de distinguer plus finement le comportement d'algorithmes mathématiquement équivalents. On peut s'en réjouir ou en avoir peur ...


λ
:=

6
Les structures de données

Où l'on apprend à mélanger les torchons et les serviettes pour définir le linge de maison.

En Caml, les types de données comprennent principalement les types somme et les types produit, c'est-à-dire les types « ou » et les types « et », encore appelés les énumérations généralisées et les enregistrements, ou plus techniquement encore l'union disjointe et les produits à champs nommés. Dans ce chapitre, nous introduisons ces différents types de données et les montrons à l'œuvre sur le problème de la représentation efficace des polynômes.

6.1 Polynômes pleins et polynômes creux

Nous avons vu par deux fois des calculs sur les polynômes, d'abord représentés par des tableaux dans le chapitre 3 (section 3.3), puis comme des listes dans le chapitre 5 (section 5.5). Nous avons appelé les seconds polynômes creux, les premiers polynômes pleins. Maintenant se pose évidemment le problème de travailler avec ces deux représentations *en même temps*, pour bénéficier des avantages de chacune d'elles : lorsqu'un polynôme est plein, la représentation à l'aide d'un tableau est économique, car les degrés sont implicites ; en revanche, lorsqu'un polynôme est creux (comporte beaucoup de coefficients nuls), la représentation en liste est préférable — quand elle n'est pas tout simplement la seule envisageable, comme pour le polynôme $1 + x^{1000000}$. Nous aimerions donc représenter les polynômes par un tableau ou une liste selon le cas, mais définir des opérations qui travaillent indifféremment sur l'une ou l'autre des représentations. Or, ces deux types de représentations sont incompatibles au point de vue du typage. Considérons la procédure d'impression des polynômes : nous avons défini deux fonctions, spécifiques à chacune des représentations, `imprime_polynôme_plein : int vect -> unit`, qui imprime les polynômes pleins, et `imprime_polynôme_creux : (int * int) list -> unit`, qui imprime les polynômes creux. Pour avoir une primitive d'impression travaillant sur tous les polynômes, on aurait donc envie d'écrire :

```
let imprime_polynôme p =
    if p «est un polynôme plein»
    then imprime_polynôme_plein p
```

```
    else imprime_polynôme_creux p;;
```

C'est effectivement la bonne idée, mais il faut la raffiner un peu : outre qu'on ne voit pas comment implémenter le prédicat « est un polynôme plein », il se pose également un problème de typage pour l'argument `p` de `imprime_polynôme` : est-ce une liste comme le suggère l'appel de fonction `imprime_polynôme_creux p`, ou un tableau pour pouvoir être passé en argument à `imprime_polynôme_plein` ? On obtiendrait forcément une erreur de typage. Par exemple, en supposant que « est un polynôme plein » renvoie toujours la valeur `true` :

```
# let imprime_polynôme p =
      if true then imprime_polynôme_plein p
      else imprime_polynôme_creux p;;
Entrée interactive:
>    else imprime_polynôme_creux p;;
>                                ^
Cette expression est de type int vect,
mais est utilisée avec le type (int * int) list.
```

Il faut donc mélanger les polynômes creux et pleins au sein d'*un même type* qui les comprenne tous les deux.

Le type `polynôme`

On définit donc un nouveau type, `polynôme`, qui établit explicitement le mélange : il indique qu'il comprend deux cas possibles, le cas des polynômes pleins qui seront des tableaux d'entiers et le cas des polynômes creux qui seront des listes de paires d'entiers.

```
# type polynôme =
     | Plein of int vect
     | Creux of (int * int) list;;
Le type polynôme est défini.
```

Le mot-clé `type` introduit la définition du nouveau type `polynôme`. Après le signe `=`, on écrit la liste des possibilités du type en cours de définition. Les noms `Plein` et `Creux` sont appelés les *constructeurs de valeurs* du type (s'il n'y a pas d'ambiguïté on dit simplement « constructeurs »). Comme d'habitude, la barre verticale `|` indique l'alternative et se lit « ou ». Le mot-clé `of` indique le type de l'argument du constructeur. Le type `polynôme` comprenant les valeurs d'un type plus les valeurs d'un autre type, on dit que c'est un type *somme*. On peut maintenant créer des valeurs de type `polynôme` en appliquant l'un des deux constructeurs du type `polynôme` à une valeur du type correspondant. Par exemple :

```
# let p1 = Plein [|1; 2; 3|];;
p1 : polynôme = Plein [|1; 2; 3|]
# let p2 = Creux [(1, 0); (1, 100)];;
p2 : polynôme = Creux [1, 0; 1, 100]
```

Maintenant `p1` et `p2` sont du même type et pourront être arguments d'une même fonction.

Le filtrage est étendu à tous les types somme et permet, étant donnée une valeur du type somme, de déterminer dans quel cas se trouve cette valeur. Pour le type `polynôme`, le filtrage va donc nous permettre d'implémenter la fonction « est un polynôme plein » :

```
# let est_un_polynôme_plein = function
    | Plein _ -> true
    | Creux _ -> false;;
est_un_polynôme_plein : polynôme -> bool = <fun>
```

Une fonction travaillant sur des valeurs de type `polynôme` fera typiquement une discrimination sur les valeurs du type par un filtrage du genre :

```
let f = function
  | Plein v -> ...
  | Creux l -> ...;;
```

Remarquez que le filtrage permet à la fois de déterminer le type du polynôme et de récupérer son tableau ou sa liste de monômes. C'est strictement analogue au cas des listes où nous écrivions :

```
let f = function
  | [] -> ...
  | x :: reste -> ...;;
```

C'est maintenant un jeu d'enfant que d'écrire la fonction d'impression des valeurs de type `polynôme` :

```
# let imprime_polynôme = function
    | Plein v -> imprime_polynôme_plein v
    | Creux l -> imprime_polynôme_creux l;;
imprime_polynôme : polynôme -> unit = <fun>
# imprime_polynôme p1;;
1 + 2x + 3x^2- : unit = ()
# imprime_polynôme p2;;
1 + x^100- : unit = ()
```

Opérations sur les valeurs de type `polynôme`

Nous définissons l'addition et la multiplication des polynômes creux ou pleins. Puisque les polynômes se présentent sous deux formes, nous avons quatre cas à envisager. L'idée est simple :

- la somme de deux polynômes creux est un polynôme creux : on appelle l'addition des polynômes creux ;
- la somme de deux polynômes pleins est un polynôme plein : on appelle l'addition des polynômes pleins ;
- la somme de deux polynômes d'espèces différentes est un polynôme creux.

En effet, si l'un des polynômes est creux il comprend beaucoup de zéros et sa somme avec un autre polynôme comprendra aussi beaucoup de zéros en général (considérez par exemple $(1 + x + 3x^2) + (1 + x^{100})$). Donc, dans le cas mixte, nous appelons encore l'addition des polynômes creux. Puisque l'un des polynômes est plein, nous avons besoin d'une fonction qui transforme un polynôme plein en polynôme creux. C'est sans

difficulté : nous parcourons le tableau des coefficients en accumulant dans une liste les monômes rencontrés. La seule subtilité est de parcourir le tableau à l'envers pour que le dernier monôme ajouté à la liste soit bien celui de degré 0.

```
# let plein_vers_creux v =
    let l = ref [] in
    for i = vect_length v - 1 downto 0 do
      if v.(i) <> 0 then l := (v.(i), i) :: !l
    done;
    !l;;
plein_vers_creux : int vect -> (int * int) list = <fun>
```

L'addition des polynômes se définit alors très simplement :

```
# let ajoute_polynômes p1 p2 =
    match p1, p2 with
    | Plein v, Plein v' -> Plein (ajoute_polynômes_pleins v v')
    | Creux l, Creux l' -> Creux (ajoute_polynômes_creux l l')
    | Plein v, Creux l ->
        Creux (ajoute_polynômes_creux (plein_vers_creux v) l)
    | Creux l, Plein v ->
        Creux (ajoute_polynômes_creux (plein_vers_creux v) l);;
ajoute_polynômes : polynôme -> polynôme -> polynôme = <fun>
```

Ce code peut être légèrement simplifié en remarquant que les deux derniers cas du filtrage sont presque identiques (ces deux cas se traduisent par deux clauses du filtrage dont la partie expression est la même). Pour éviter cette redite, on joue sur le fait que l'addition des polynômes est commutative pour traiter le dernier cas par un appel récursif à la fonction `ajoute_polynôme` qui inverse les arguments `p1` et `p2`.

```
# let rec ajoute_polynômes p1 p2 =
    match p1, p2 with
    | Plein v, Plein v' -> Plein (ajoute_polynômes_pleins v v')
    | Creux l, Creux l' -> Creux (ajoute_polynômes_creux l l')
    | Plein v, Creux l ->
        Creux (ajoute_polynômes_creux (plein_vers_creux v) l)
    | Creux l, Plein v ->
        ajoute_polynômes p2 p1;;
ajoute_polynômes : polynôme -> polynôme -> polynôme = <fun>
```

Cette dernière solution permet de ne pas dupliquer de code, ce qui raccourcit légèrement le texte de la fonction et diminue la probabilité d'introduire une erreur en ne modifiant qu'une des clauses lors de corrections ultérieures du programme. En fait, lorsque l'expression à renvoyer est compliquée, l'appel récursif s'impose sans contestation possible. Cependant, cette solution présente l'inconvénient de suggérer que la fonction `ajoute_polynôme` est vraiment récursive, alors qu'elle ne l'est que pour des raisons « administratives ».

La multiplication n'est pas plus compliquée :

```
# let rec multiplie_polynômes p1 p2 =
    match p1, p2 with
    | Plein v, Plein v' -> Plein (multiplie_polynômes_pleins v v')
    | Creux l, Creux l' -> Creux (multiplie_polynômes_creux l l')
    | Plein v, Creux l ->
```

```
        Creux (multiplie_polynômes_creux (plein_vers_creux v) l)
    | Creux l, Plein v ->
        multiplie_polynômes p2 p1;;
multiplie_polynômes : polynôme -> polynôme -> polynôme = <fun>
# imprime_polynôme (multiplie_polynômes p1 p2);;
1 + 2x + 3x^2 + x^100 + 2x^101 + 3x^102- : unit = ()
# let p10000 = Creux [(1, 0); (1, 10000)];;
p10000 : polynôme = Creux [1, 0; 1, 10000]
# imprime_polynôme (multiplie_polynômes p10000 p10000);;
1 + 2x^10000 + x^20000- : unit = ()
```

6.2 Types sommes élaborés

Un autre exemple classique de type somme est la modélisation des peintures. On suppose que les peintures sont décrites soit par un nom explicite, soit par un simple numéro de référence, soit par un mélange d'autres peintures. Nous envisagerons successivement ces trois cas et construirons donc le type `peinture` par raffinements successifs, en trois étapes.

Énumérations

On considère d'abord les peintures explicitement nommées, en supposant qu'il en existe trois : le `Bleu`, le `Blanc` et le `Rouge`. Le type `peinture` comporte donc trois cas : c'est un type somme. Ces cas ne sont plus des valeurs de types différents comme pour les polynômes, mais simplement trois constantes. On les modélise par trois constructeurs *sans arguments*, donc sans partie `of` dans la définition :

```
# type peinture =
    | Bleu
    | Blanc
    | Rouge;;
Le type peinture est défini.
```

Les trois constructeurs sont maintenant trois nouvelles constantes du langage Caml, de type `peinture`.

```
# let p = Bleu;;
p : peinture = Bleu
```

Tout naturellement, le filtrage s'applique aussi à ce nouveau type :

```
# let est_blanche = function
    | Blanc -> true
    | _ -> false;;
est_blanche : peinture -> bool = <fun>
# est_blanche p;;
- : bool = false
```

Ces types somme ne comportant que des constantes sont appelés *types énumérés*. Vous en connaissez déjà : par exemple, le type `bool` est un type somme énuméré à deux constantes, `true` et `false`.

Types à constructeurs non constants

Nous supposons maintenant qu'il existe dans l'ensemble de toutes les peintures des teintes qui n'ont pas de nom, mais seulement un numéro de référence. Nous étendons donc le type `peinture` avec un nouveau constructeur qui prenne en compte ce cas. Il s'agit maintenant d'un constructeur ayant un argument : le numéro de référence. Appelons ce constructeur `Numéro`. Par exemple, `Numéro 14` modélisera la peinture de référence numéro 14. Nous définissons donc le nouveau type des peintures comme :

```
# type peinture =
     | Bleu
     | Blanc
     | Rouge
     | Numéro of int;;
Le type peinture est défini.
```

Types récursifs

La prochaine étape est la description des mélanges de peintures. Il existe maintenant des peintures qui sont simplement des mélanges de deux autres peintures (en proportions égales) et qu'on identifie par les peintures qui les composent. Nous introduisons donc un nouveau constructeur `Mélange` avec pour argument un couple de peintures. Notre type devient :

```
# type peinture =
     | Bleu
     | Blanc
     | Rouge
     | Numéro of int
     | Mélange of peinture * peinture;;
Le type peinture est défini.
# let mél1 = Mélange (Bleu, Blanc);;
mél1 : peinture = Mélange (Bleu, Blanc)
# let mél2 = Mélange (Numéro 0, Rouge);;
mél2 : peinture = Mélange (Numéro 0, Rouge)
```

Remarquez que le type `peinture` est devenu récursif, puisqu'il intervient dans sa propre définition. Ainsi, on peut mélanger n'importe quelles peintures et en particulier faire des mélanges de plus de deux peintures.

```
# let mél3 = Mélange (mél1,mél2);;
mél3 : peinture =
 Mélange (Mélange (Bleu, Blanc), Mélange (Numéro 0, Rouge))
```

Le filtrage sur le type `peinture` ne pose pas de problèmes :

```
# let rec contient_du_bleu = function
    | Bleu -> true
    | Mélange (p1,p2) -> contient_du_bleu p1 || contient_du_bleu p2
    | _ -> false;;
contient_du_bleu : peinture -> bool = <fun>
# contient_du_bleu mél3;;
- : bool = true
```

La définition du type `peinture`, quoique récursive, conserve tout de même un sens, parce qu'il existe des cas de base pour arrêter la récursion. C'est tout à fait analogue aux définitions de fonctions récursives qui présentent des cas d'arrêt simples. Les cas de base du type, comme par exemple les constructeurs sans arguments, correspondent souvent à des cas de base des fonctions récursives sur ce type.

Les cartes

On modélise très aisément un jeu de cartes en utilisant les types somme. Les couleurs forment un type énuméré :

```
# type couleur = | Trèfle | Carreau | Coeur | Pique;;
Le type couleur est défini.
```

et les cartes un type somme à plusieurs possibilités, selon les valeurs faciales des cartes :

```
# type carte =
     | As of couleur
     | Roi of couleur
     | Dame of couleur
     | Valet of couleur
     | Petite_carte of int * couleur;;
Le type carte est défini.
```

Dans cette définition, nous avons choisi de regrouper toutes les cartes qui ne sont pas des figures sous la même dénomination : `Petite_carte`. On aurait pu aussi continuer l'énumération avec des constructeurs `Dix`, `Neuf`, `Huit`, etc.

Pour illustrer le filtrage sur les types somme, nous définissons la valeur d'une carte à la « belote ». Cette valeur dépend d'une couleur particulière, l'atout, choisie par les joueurs à chaque tour. Les cartes dont la valeur change sont le valet et le neuf : le neuf compte d'ordinaire pour 0, mais vaut 14 quand il est de la couleur de l'atout, et le valet d'atout vaut 20 au lieu de 2 d'ordinaire. D'autre part, les dix valent 10 points et les autres petites cartes 0.

```
# let valeur_d'une_carte couleur_d'atout = function
    | As _ -> 11
    | Roi _ -> 4
    | Dame _ -> 3
    | Valet c -> if c = couleur_d'atout then 20 else 2
    | Petite_carte (10, _) -> 10
    | Petite_carte (9, c) -> if c = couleur_d'atout then 14 else 0
    | _ -> 0;;
valeur_d'une_carte : couleur -> carte -> int = <fun>
```

Remarquez que la structure du filtrage de la fonction `valeur_d'une_carte` est très similaire à la définition du type `carte`. C'est un mécanisme fréquent en Caml : pour définir une fonction sur un type somme, on se guide souvent sur la définition du type qui donne le squelette du filtrage à utiliser. On le complète alors pour envisager les cas particuliers, comme ici les cas du 10 et du 9.

Cela termine les exemples de types somme. Nous donnons maintenant une présentation plus générale du concept.

6.3 Les types somme

Les types somme servent donc à modéliser des données comprenant des alternatives. On les appelle aussi types « ou », car une donnée modélisée par un type somme est d'une espèce *ou* d'une autre *ou* ... Par exemple, une peinture est soit nommée, soit un simple numéro, soit un mélange de deux peintures ; de même, les polynômes sont soit pleins, soit creux.

Les fonctions définies sur un type somme opèrent généralement par filtrage sur les éléments du type ; elles ont une structure très voisine de la définition du type. On peut considérer qu'on emploie alors une programmation « dirigée par le filtrage » (ou dirigée par les définitions de type).

Remarquons qu'il est d'usage de toujours mettre une majuscule aux noms des constructeurs de type somme, pour ne pas les confondre avec les noms de variables dans le filtrage. Corrélativement, il est recommandé de toujours écrire les noms de variables en minuscules. Cette règle n'est pas absolue : par exemple les booléens `true` et `false` dérogent à cette règle. Il est vrai que leur statut est très particulier puisque ce sont des constructeurs primitifs et des mots-clés du langage.

Formellement, les types somme sont l'analogue Caml de la notion mathématique de *somme disjointe d'ensemble*. Nous n'étudierons pas cette notion, nous contentant de l'idée intuitive ci-dessus.

6.4 Les types produit

Caml offre une deuxième classe de structures de données, complémentaires des types somme : les types produit, encore appelés *enregistrements* ou *records* en anglais.

Définition des types enregistrement

Les enregistrements constituent une généralisation des n-uplets. Ils servent à modéliser les données qui ont simultanément plusieurs propriétés. On les appelle donc aussi types « et », car ils s'utilisent pour représenter une donnée ayant telle caractéristique *et* telle autre caractéristique *et* ... On dresse alors la liste des caractéristiques lors de la définition du type. Chacune des caractéristiques est accompagnée de son type respectif et la liste de toutes les caractéristiques est entourée d'accolades `{ }`. Par exemple, un monôme possède un coefficient *et* un degré ; pour modéliser les monômes, nous définirons donc un type produit avec deux caractéristiques, `coefficient` et `degré`, toutes deux de type entier.

```
# type monôme = { coefficient : int; degré : int };;
Le type monôme est défini.
```

Pour construire une valeur de ce type, on se contente d'énumérer ses caractéristiques particulières :

```
# let m1 = {coefficient = 1; degré = 0};;
m1 : monôme = {coefficient = 1; degré = 0}
```

Accès aux composantes des enregistrements

Pour accéder à l'une des caractéristiques d'un objet de type produit, il suffit de faire suivre l'objet d'un point « . » et du nom de la caractéristique. C'est analogue au « . » de l'accès dans les tableaux et les chaînes de caractères.

```
# m1.coefficient;;
- : int = 1
# m1.degré;;
- : int = 0
```

Ce que nous avons appelé « nom de caractéristique » porte le nom technique d'*étiquette*. Une caractéristique s'appelle aussi une *rubrique*, ou encore un *champ* de l'enregistrement. Les étiquettes permettent de se libérer de l'ordre dans lequel on nomme les caractéristiques : le compilateur se charge de les mettre dans le bon ordre. Ainsi, si l'on intervertit l'ordre des étiquettes, la valeur produite est la même :

```
# let m2 = { degré = 0; coefficient = 1 };;
m2 : monôme = {coefficient = 1; degré = 0}
# m1 = m2;;
- : bool = true
```

Filtrage des types enregistrement

Le filtrage s'étend naturellement aux enregistrements :

```
# let de_degré_zéro = function
    | {degré = 0; coefficient = _} -> true
    | _ -> false;;
de_degré_zéro : monôme -> bool = <fun>
# de_degré_zéro m1;;
- : bool = true
```

Il existe en plus une ellipse spéciale pour indiquer qu'on ne s'intéresse pas au reste des champs d'un enregistrement : on écrit « `; _` » pour dire « quels que soient les autres champs et leurs contenus ». On écrirait donc la fonction `de_degré_zéro` plus naturellement ainsi :

```
# let de_degré_zéro = function
    | {degré = 0; _} -> true
    | _ -> false;;
de_degré_zéro : monôme -> bool = <fun>
```

On peut aussi nommer le contenu des champs et faire des synonymes. Par exemple, la fonction qui renvoie la valeur absolue d'un monôme s'écrit :

```
# let abs_monôme = function
      {coefficient = a; degré = d} as m ->
        if a < 0 then {coefficient = -a; degré = d} else m;;
abs_monôme : monôme -> monôme = <fun>
```

Les types enregistrement portent le nom technique de *produits nommés à champs nommés*, et les types produit en général correspondent à la notion mathématique de *produit cartésien d'ensembles*. Encore une fois, nous nous contenterons de la vision intuitive.

6.5 Mélange de types somme et types produit

Types somme et types produit peuvent être arbitrairement mélangés. Ainsi, une définition plus élaborée des polynômes serait :

```
# type poly =
    | Plein of int vect
    | Creux of monôme list;;
Le type poly est défini.
```

L'avantage de cette définition est d'avoir explicitement nommé les caractéristiques d'un monôme. Par exemple, la fonction `plein_vers_creux` deviendrait :

```
# let plein_vers_creux v =
     let l = ref [] in
     for i = vect_length v - 1 downto 0 do
      l := {degré = i; coefficient = v.(i)} :: !l
     done;
     !l;;
plein_vers_creux : int vect -> monôme list = <fun>
```

On a donc remplacé la paire `(v.(i), i)` par l'enregistrement `{degré = i; coefficient = v.(i)}` dans le code de l'ancienne fonction. La différence est faible mais elle suffit à rendre le code plus facile à lire et à écrire. D'ailleurs la première version de `plein_vers_creux` écrite au cours de la rédaction de ce livre était fausse : au lieu de `(v.(i), i)` nous avions écrit `(i, v.(i))` parce que nous avions oublié dans quel ordre nous avions décidé de mettre le degré et le coefficient dans les couples d'entiers modélisant les monômes ! Cela ne se voyait pas sur le type des polynômes creux, la convention étant simplement répartie dans le code des fonctions de manipulation des polynômes creux. Cette erreur, indétectable par typage, est facilement évitée quand on écrit explicitement les noms `degré` et `coefficient` et que c'est le compilateur qui rétablit l'ordre des rubriques.

6.6 Structures de données mutables

Nous connaissons déjà certaines structures de données dont le contenu est modifiable dynamiquement : ce sont les références et les tableaux. Les enregistrements possèdent aussi cette qualité : lors de la définition d'un type enregistrement, certains champs peuvent être qualifiés de « mutables », c'est-à-dire modifiables. Le vérificateur de type autorise alors la modification physique du contenu du champ des objets de ce type.

Enregistrements à champs mutables

Voici un exemple très simple où il est nécessaire de modifier des champs d'enregistrement : supposez qu'on veuille modéliser des comptes bancaires. En première approximation, un compte se caractérise par son numéro et par le montant actuel des dépôts sur le compte (le solde du compte). Si le numéro du compte a peu de chance de changer, en revanche le solde varie à chaque retrait ou dépôt. Il faut donc que l'étiquette `solde` du type `compte` soit déclarée `mutable` à la définition du type.

```
# type compte = { numéro : int; mutable solde : float };;
Le type compte est défini.
```

La définition d'une valeur d'un type enregistrement à champs mutables et l'accès à ses champs ne diffèrent en rien du cas d'un type enregistrement normal.

```
# let compte_de_durand = {numéro = 0; solde = 1000.0};;
compte_de_durand : compte = {numéro = 0; solde = 1000.0}
# compte_de_durand.solde;;
- : float = 1000.0
```

Pour définir la fonction `dépôt` qui met à jour le solde d'un compte lors d'un dépôt, on utilise l'opération de modification physique d'un champ d'enregistrement, notée e_1.*étiquette* `<-` e_2, où e_1 et e_2 sont deux expressions et *étiquette* le nom d'une étiquette d'enregistrement. L'exécution de cette expression remplace le contenu du champ *étiquette* de l'enregistrement e_1 par la valeur de l'expression e_2. Comme toute modification physique, cette expression renvoie la valeur « rien » `()`. La fonction `dépôt` s'écrit donc simplement :

```
# let dépôt compte montant =
      compte.solde <- montant +. compte.solde;;
dépôt : compte -> float -> unit = <fun>
# dépôt compte_de_durand 30.0;;
- : unit = ()
# compte_de_durand.solde;;
- : float = 1030.0
```

Variables rémanentes

Cet exemple nous permet aussi d'aborder les références locales aux fonctions qui conservent leur valeur entre les différents appel à la fonction. Nous définissons la fonction de création des comptes : elle doit nécessairement allouer un nouveau numéro à chaque nouveau compte créé. Pour cela, il lui suffit de tenir à jour une référence entière contenant le dernier numéro de compte attribué et de l'incrémenter à chaque création. Pour s'assurer que cette référence ne peut être modifiée par inadvertance dans une autre partie du programme de gestion des comptes, on la rend complètement locale au corps de la fonction qui crée les comptes :

```
# let crée_compte =
      let numéro_de_compte = ref 0 in
      (function dépot ->
        numéro_de_compte := !numéro_de_compte + 1;
        {numéro = !numéro_de_compte; solde = dépot});;
crée_compte : float -> compte = <fun>
# let compte_de_dupont = crée_compte 500.0;;
compte_de_dupont : compte = {numéro = 1; solde = 500.0}
# let compte_de_duval = crée_compte 1000.0;;
compte_de_duval : compte = {numéro = 2; solde = 1000.0}
```

Il faut bien comprendre que la référence `numéro_de_compte` est créée une seule fois, lors de la construction de la fonction `crée_compte`. À chaque appel de `crée_compte`, on retrouve ainsi dans `numéro_de_compte` la dernière valeur qui y a été inscrite. Les

variables de ce genre sont appelées *variables rémanentes* dans la littérature informatique (et variables *statiques* en C). Remarquez que Caml les autorise, sans avoir besoin de fournir une construction spéciale pour cela : le `let in` habituel et les fonctions anonymes suffisent pour programmer des variables rémanentes.

6.7 Structures de données et filtrage

Comme nous l'avons vu à maintes reprises, le filtrage va de paire avec les définitions de structures de données. Bien plus, la définition d'un type sert de guide pour écrire le squelette du filtrage des fonctions qui opèrent sur ce type. Nous voulons cependant attirer votre attention sur quelques traits avancés du filtrage et quelques écueils qui guettent les débutants qui écrivent leurs premiers filtrages.

Filtrage de valeurs calculées

En premier lieu, il faut conserver à l'esprit que le filtrage en Caml est *structurel* : on ne peut utiliser dans les motifs que des constructeurs, des constantes et des variables, à l'exclusion des valeurs calculées. C'est pourquoi les variables qui interviennent dans un motif ne servent jamais à faire des tests, mais au contraire à lier des parties de la valeur filtrée. Comparer par exemple, la définition (erronée) de la fonction `est_un` avec celle (correcte) de la fonction `test_à_un` :

```
# let un = 1;;
un : int = 1
# let est_un = function
    | un -> true
    | _ -> false;;
Entrée interactive:
>  | _ -> false;;
>    ^
Attention: ce cas de filtrage est inutile.
est_un : 'a -> bool = <fun>
# est_un 2;;
- : bool = true
# let test_à_un x = if x = un then true else false;;
test_à_un : int -> bool = <fun>
# test_à_un 2;;
- : bool = false
```

Le premier filtre de la fonction `est_un` comprend la variable `un`, qui est sans rapport avec l'identificateur `un` précédemment défini à la valeur 1. Autrement dit, le nom de la variable `un` est sans importance dans le filtrage de la fonction `est_un` : on peut le remplacer par `x` ou `y`, et le filtrage est équivalent à `x -> true | _ -> false`. Contrairement à ce que l'utilisateur voulait sans doute exprimer, la fonction `est_un` ne teste donc pas si son argument correspond à la même valeur que l'identificateur `un` : en fait, la fonction `est_un` renvoie toujours `true`. Cela explique le résultat de `est_un 2`. Cela explique aussi le message du compilateur : « `ce cas de filtrage est inutile` ». Le compilateur s'est rendu compte que le cas `_ ->` ne servira jamais. C'est

pour éviter ce genre de confusions qu'on utilise la convention de faire commencer les noms de constructeurs par une majuscule et d'écrire les variables dans les filtres en minuscules. Retenons que

> Toute variable dans un filtre est une nouvelle variable.

Linéarité du filtrage

Il faut également savoir que le filtrage en Caml est *linéaire*, ce qui signifie qu'un nom de variable ne peut apparaître qu'une seule fois dans un filtre. Cette contrainte est violée le plus souvent lorsqu'on veut tester l'égalité de deux morceaux d'une valeur. Voici une tentative (erronée) de définir la fonction d'égalité : si le couple argument comporte deux composantes identiques on renvoie vrai et sinon on renvoie faux.

```
# let égal = function
    | (x, x) -> true
    | _ -> false;;
Entrée interactive:
>  | (x, x) -> true
>         ^
L'identificateur x est défini plusieurs fois dans ce motif.
```

Les tests d'égalité opérés par le filtrage ne concernent que les constantes (les constructeurs). Les tests d'égalité plus généraux ne s'expriment pas par filtrage, ils doivent faire l'objet d'une alternative explicite (un `if then else`) dans l'expression d'une clause de filtrage ou bien d'une *garde* que nous étudions brièvement dans le prochain paragraphe. Une définition acceptable de `égal` serait donc :

```
# let égal = function (x, y) -> if x = y then true else false;;
égal : 'a * 'a -> bool = <fun>
```

Remarque : comme nous l'avons vu au paragraphe 2.1, l'alternative du corps de `égal` est inutile ; on la remplace donc simplement par sa partie condition pour obtenir

```
# let égal (x, y) = x = y;;
égal : 'a * 'a -> bool = <fun>
```

Ce qui nous permet de constater que la fonction `égal` ne définit pas la fonction d'égalité mais est en fait un synonyme de l'opérateur `=` (plus précisément `égal` est la version non curryfiée de l'opérateur `=`).

Combiner filtrage et tests : les gardes

Vous remarquerez sans doute que le filtrage permet une programmation particulièrement claire, et autorise l'écriture compacte de multiples conditions. Cependant le simple filtrage structurel (et linéaire) ne permet pas de mélanger la sélection sur la forme de la valeur filtrée et les tests sur les valeurs effectives des composants du filtre. C'est pourquoi le filtrage de Caml propose une construction supplémentaire, les *gardes*, pour effectuer des tests arbitraires pendant le filtrage. La clause

`|` *filtre* `when` *condition* `->` ...

où `condition` est une expression booléenne quelconque, filtre les mêmes valeurs que `filtre`, mais elle n'est sélectionnée que dans le cas où `condition` est vraie ; dans le cas contraire le filtrage continue normalement en séquence.

À l'aide d'une garde, on écrit facilement une version correcte de la fonction `est_un` :

```
# let est_un = function
    | x when x = un -> true
    | _ -> false;;
est_un : int -> bool = <fun>
# est_un 2;;
- : bool = false
```

On écrit aussi la fonction `valeur_d'une_carte` encore plus élégamment :

```
# let valeur_d'une_carte couleur_d'atout = function
    | As _ -> 11
    | Roi _ -> 4
    | Dame _ -> 3
    | Valet c when c = couleur_d'atout -> 20
    | Valet _ -> 2
    | Petite_carte (9, c) when c = couleur_d'atout -> 14
    | Petite_carte (10, _) -> 10
    | _ -> 0;;
valeur_d'une_carte : couleur -> carte -> int = <fun>
```

Filtrage exhaustif, filtrage partiel

Enfin, il faut se méfier des filtrages *non exhaustifs*, c'est-à-dire des filtrages qui oublient des cas. C'est une des forces du filtrage de servir de guide pour facilement envisager tous les cas concernant une structure de données, il faut donc en profiter pleinement. Si vous oubliez d'envisager certains cas, le compilateur émet un message d'avertissement et il faut en tenir compte. Voici un exemple caricatural de filtrage non exhaustif :

```
# let vide = function [] -> true;;
Entrée interactive:
>let vide = function [] -> true;;
>            ^^^^^^^^^^^^^^^^^^
Attention: ce filtrage n'est pas exhaustif.
vide : 'a list -> bool = <fun>
```

Dans cette situation, il faut vous efforcer de « boucher les trous » de votre filtrage. Laisser des filtrages non exhaustifs dans un programme est généralement considéré comme un laisser-aller de mauvais aloi.

6.8 Structures de données et récurrence

Cette section peut être sautée en première lecture. Nous y étendons le principe de récurrence aux types de données.

Comme pour les listes, nous aimerions disposer d'un moyen de prouver des propriétés sur les programmes qui utilisent des types somme ou des types produit. Comme

dans le cas des listes, il suffit de trouver un nombre lié aux données, sur lequel on puisse employer le principe de récurrence. Pour les listes, nous avions utilisé la longueur. En fait, il suffit pour généraliser à tous les types somme de remarquer que la longueur d'une liste n'est autre que le nombre d'utilisation du constructeur « `::` » nécessaire à la construction de la liste. Pour un type somme général, on pourra ainsi raisonner sur le nombre d'occurrences des constructeurs du type. Le cas particulier de la liste vide sera transposé à tous les constructeurs constants du type, tandis que le cas de « `::` » correspondra au nombre d'occurrences des constructeurs non constants.

Pour les types produit, il y a plusieurs manières d'étendre le raisonnement par récurrence. Pour simplifier, supposons que nous devions démontrer une propriété sur un type produit à deux composantes seulement et pour simplifier encore supposons que ce soit le type `int * int`. Pour prouver une propriété P qui dépend d'une paire d'entiers (m, n), on utilisera par exemple le principe de récurrence suivant :

> Si $P(0,0)$ est vraie, et dès que $P(m,n)$ est vraie alors $P(m+1,n)$ et $P(m,n+1)$ sont vraies, alors P est vraie pour tous m et n.

Ce principe se démontre facilement à partir du principe de base et se généralise sans difficulté à un nombre quelconque de variables.

Cette extension du principe de récurrence aux structures de données se nomme tout naturellement la récurrence structurelle. C'est le principal moyen de démonstration à la disposition de l'informaticien. Il est heureux qu'il soit si simple à appréhender.

7
Le docteur

Où l'intelligence naturelle vient au secours de la bêtise artificielle.

On vous initie ici au maniement des exceptions, des listes d'association et des entrées-sorties élémentaires de Caml. Par ailleurs, on implémente un programme qui essaye de se faire passer pour un psychanalyste ...

7.1 Vue d'ensemble

Notre « psychanalyste » électronique s'appelle Camélia. C'est un programme interactif, qui pose des questions à son « patient » et réagit à ses réponses en imprimant un message au terminal. La stratégie du programme repose donc sur l'examen des phrases entrées par le patient. Une fois la phrase examinée, le programme choisit au hasard une réponse parmi un ensemble de réponses toutes faites, préparées par le programmeur. L'examen d'une phrase consiste simplement à chercher des mots connus du programme dans la phrase qui lui est soumise. Les phrases sont classées sommairement en trois catégories : les phrases très simples (par exemple réduites à « oui » ou « non »), pour lesquelles Camélia dispose d'un ensemble de réponses types ; les questions (phrases terminées par un caractère « `?` ») auxquelles Camélia répond « C'est moi qui pose les questions » ou une phrase du même genre ; et les phrases complexes, pour lesquelles Camélia cherche un mot intéressant (comme « famille » ou « père » ou « sexe »). Si le programme trouve un mot intéressant, il réagit par une phrase type (toujours tirée au hasard parmi un ensemble préparé). S'il ne trouve pas de mot intéressant, Camélia n'a rien à dire d'intéressant non plus : elle se contente de relancer la conversation, en imprimant par exemple « Parlez-moi un peu de vous ». Tout le sel de la conversation avec Camélia vient de l'utilisateur. C'est lui qui attache un sens précis aux phrases creuses de Camélia. L'art du programmeur ne consiste ici qu'à faire dire au programme les phrases les plus creuses et les plus ambiguës possible, pour laisser l'utilisateur interpréter lui-même.

Pour implémenter Camélia, nous aurons besoin de fonctions nouvelles sur les listes et les chaînes de caractères. D'autre part nous aurons besoin aussi d'introduire le mécanisme d'*exceptions* de Caml. En effet, le cœur du programme consiste à essayer

successivement les différentes stratégies possibles, jusqu'à ce que l'une d'entre elles réussisse. Nous verrons que c'est justement un emploi typique des exceptions.

7.2 Les exceptions

Erreurs et rattrapage d'erreurs

Dans les langages fonctionnels, toute fonction qui ne boucle pas indéfiniment doit rendre une valeur, quel que soit son argument. Malheureusement certaines fonctions, bien que nécessairement définies pour toute valeur de leur type argument, ne peuvent pas retourner de valeur sensée pour tous les arguments possibles. Considérez par exemple la division entre nombres entiers : que doit-elle faire lorsqu'on tente de diviser par 0 ? Le problème se pose aussi pour les données structurées : considérez la fonction `tête` qui renvoie la tête d'une liste. Que peut-elle faire lorsque son argument est la liste vide ? Dans de telles situations la fonction doit échouer, c'est-à-dire arrêter les calculs et signaler une erreur. C'est ce que nous avons fait en utilisant la fonction prédéfinie `failwith` :

```
# failwith;;
- : string -> 'a = <fun>
```

qui envoie un message indiquant la cause de l'échec. C'est pourquoi nous définissons `tête` par :

```
# let tête = function
    | [] -> failwith "tête"
    | x::_ -> x;;
tête : 'a list -> 'a = <fun>
```

Et maintenant, `tête []` nous signale une erreur dans la fonction `tête` :

```
# tête [];;
Exception non rattrapée: Failure "tête"
```

Ce mécanisme de déclenchement d'erreurs est utile, mais il se peut que nous voulions récupérer ces erreurs, parce que nous savons comment continuer les calculs après une telle erreur (qui devient une erreur « attendue » du point de vue du programmeur). Par exemple, imaginons qu'on doive ajouter systématiquement la tête d'une liste à un compteur. Si la liste est vide, il est logique de continuer les calculs en n'ajoutant rien au compteur. Dans ce cas, l'échec signalé par la fonction `tête` doit être récupéré. On utilise pour cela la construction `try` ... `with` ... (*try* signifie essayer et *with* avec) qui permet de calculer une expression en surveillant les exceptions que son calcul peut déclencher. Cette construction obéit à la syntaxe suivante : `try` *expression* `with` *filtrage*. Elle signifie intuitivement : essayer de calculer la valeur de *expression* et si cette évaluation déclenche une erreur qui tombe dans un des cas du *filtrage* alors retourner la valeur correspondante de la clause sélectionnée par le filtrage. Par exemple, puisque l'erreur signalée par la fonction `tête` est `Failure "tête"`, on envisagera cet échec dans la partie filtrage du `try` ... `with` ... pour renvoyer une valeur entière, comme si aucune erreur n'avait été déclenchée. On rattrape donc l'échec sur la liste vide et l'on renvoie 0, par la phrase :

```
# try (tête []) with Failure "tête" -> 0;;
- : int = 0
```

On écrira donc la procédure d'incrémentation du compteur :

```
# let ajoute_au_compteur compteur l =
  compteur := !compteur + (try (tête l) with Failure "tête" -> 0);;
ajoute_au_compteur : int ref -> int list -> unit = <fun>
# let c = ref 0;;
c : int ref = ref 0
# ajoute_au_compteur c [1]; !c;;
- : int = 1
# ajoute_au_compteur c []; !c;;
- : int = 1
```

C'est la méthode élémentaire d'utilisation des exceptions de Caml. Nous décrivons maintenant le mécanisme dans toute sa généralité.

Valeurs exceptionnelles

Le trait distinctif du traitement d'erreurs en Caml, et ce qui en fait la généralité, est le statut des erreurs : ce sont des valeurs à part entière du langage. Elles appartiennent à un type prédéfini `exn` et on les appelle « valeurs exceptionnelles ». On les manipule donc comme toutes les autres valeurs. Par exemple, l'échec signalé par la fonction `tête` est la valeur exceptionnelle `Failure "tête"` :

```
# let échec_de_tête = Failure "tête";;
échec_de_tête : exn = Failure "tête"
```

Remarquez que `Failure` n'est rien d'autre qu'un constructeur de valeurs du type `exn`.

La propriété caractéristique des valeurs exceptionnelles est évidemment qu'on peut les déclencher (on dit aussi les lever, par analogie avec la locution « lever une perdrix »). Pour lever une valeur exceptionnelle on utilise la fonction prédéfinie `raise` (en anglais *to raise* signifie « lever ») :

```
# raise;;
- : exn -> 'a = <fun>
# raise échec_de_tête;;
Exception non rattrapée: Failure "tête"
```

La primitive `raise` est une fonction « magique », car elle n'est pas définissable dans le langage. Elle interrompt immédiatement les calculs en cours pour déclencher le signal (lever la valeur exceptionnelle) qu'elle a reçu en argument. C'est ce qui explique qu'un appel à la fonction `raise` puisse intervenir dans n'importe quel contexte avec n'importe quel type : les calculs ne seront de toute façon jamais effectués lorsqu'on évaluera l'appel à `raise`, le contexte peut donc faire toutes les hypothèses qu'il désire sur la valeur renvoyée par `raise`. Par exemple :

```
# 1 + (raise échec_de_tête);;
Exception non rattrapée: Failure "tête"
# "Bonjour" ^ (raise échec_de_tête);;
Exception non rattrapée: Failure "tête"
```

Bien entendu, les phrases essentiellement mal typées, où `raise` apparaît dans un contexte lui-même mal typé, sont toujours rejetées :

```
# 1 + (raise échec_de_tête) ^ "Bonjour";;
Entrée interactive:
>1 + (raise échec_de_tête) ^ "Bonjour";;
>^^^^^^^^^^^^^^^^^^^^^^^^
Cette expression est de type int,
mais est utilisée avec le type string.
```

La construction `try ... with`

On peut donc considérer les valeurs exceptionnelles comme des signaux qu'on envoie à l'aide de la fonction `raise` et qu'on reçoit avec la construction `try` ... `with` ... La sémantique de `try` *e* `with` *filtrage* est de retourner la valeur de *e* si *e* s'évalue « normalement », c'est-à-dire sans déclenchement d'exception. En revanche, si une valeur exceptionnelle est déclenchée pendant l'évaluation de *e*, alors cette valeur est filtrée avec les clauses du *filtrage* et comme d'habitude la partie *expression* de la clause sélectionnée est renvoyée. Ainsi, la partie filtrage de la construction `try` ... `with` ... est un filtrage parfaitement ordinaire, opérant sur des valeurs du type `exn`. La seule différence est qu'en cas d'échec du filtrage, la valeur exceptionnelle qu'on n'a pas pu filtrer est propagée, c'est-à-dire déclenchée à nouveau. Comparez ainsi une évaluation habituelle :

```
# try tête [1] with Failure "tête" -> 0;;
- : int = 1
```

une évaluation déclenchant une valeur exceptionnelle rattrapée :

```
# try tête [] with Failure "tête" -> 0;;
- : int = 0
```

et finalement une propagation de valeur exceptionnelle :

```
# try tête [] with Failure "reste" -> 0;;
Exception non rattrapée: Failure "tête"
```

Définition d'exceptions

De nombreuses fonctions prédéfinies de Caml, quand elles échouent, déclenchent l'exception `Failure` avec leur nom comme argument. C'est pourquoi l'exception `Failure` possède un « déclencheur » prédéfini, la fonction `failwith`. Nous pouvons maintenant écrire sa définition :

```
# let failwith s = raise (Failure s);;
failwith : string -> 'a = <fun>
```

Si les exceptions prédéfinies ne vous satisfont pas, parce que vous souhaitez par exemple que votre valeur exceptionnelle transporte autre chose qu'une chaîne de caractères, vous pouvez définir une nouvelle exception. En effet, le type `exn` est un type somme (il y a plusieurs exceptions différentes ; c'est donc un type « ou »), mais d'un genre très particulier : sa définition n'est jamais achevée. C'est pourquoi il est possible à tout moment de lui ajouter de nouveaux constructeurs, soit constants soit fonctionnels. Pour définir un nouveau constructeur du type `exn`, donc une nouvelle exception, on utilise le mot-clé `exception` suivi d'une définition de constructeur de type somme. Pour définir la nouvelle exception constante `Stop`, on écrira donc simplement :

```
# exception Stop;;
L'exception Stop est définie.
```

La définition d'une exception fonctionnelle comportera une partie « `of` *type* » qui précise le type de l'argument de l'exception.

```
# exception Erreur_fatale of string;;
L'exception Erreur_fatale est définie.
# raise (Erreur_fatale "Cas imprévu dans le compilateur");;
Exception non rattrapée: Erreur_fatale "Cas imprévu dans le compilateur"
```

Voici la description précise des définitions d'exception à l'aide de diagrammes syntaxiques :

Définition d'exceptions	::=	`exception` *définition-de-constructeur* (`and` *définition-de-constructeur*)*
définition-de-constructeur	::=	*identificateur*
	|	*identificateur* `of` *type*

Les exceptions comme moyen de calcul

Les exceptions ne servent pas seulement à gérer les erreurs : elles sont aussi utilisées pour calculer. Dans ce cas, la valeur exceptionnelle transporte un résultat, ou bien signale un événement attendu. À titre démonstratif, nous définissons la fonction `caractère_dans_chaîne`, qui teste l'appartenance d'un caractère à une chaîne et dont nous avons besoin pour implémenter Camélia. On pourrait évidemment écrire cette fonction à l'aide d'une fonction récursive locale :

```
# let caractère_dans_chaîne chaîne car =
      let rec car_dans_chaîne i =
          i < string_length chaîne &&
          (chaîne.[i] = car ||
           car_dans_chaîne (i + 1)) in
      car_dans_chaîne 0;;
caractère_dans_chaîne : string -> char -> bool = <fun>
```

Cependant, cette fonction récursive code évidemment une boucle ; nous préférons donc l'écrire avec une boucle. On parcourt donc la chaîne argument à l'aide d'une boucle `for` en recherchant le caractère donné. Cependant, que faire si le caractère est trouvé ? Il faut arrêter la boucle et signaler sa présence. Ce comportement revient à déclencher une exception. Nous définissons donc l'exception `Trouvé`. Et nous surveillons la boucle de recherche : si l'exception est déclenchée, la fonction renvoie `true`. En revanche, si la boucle se termine normalement, c'est que le caractère n'était pas dans la chaîne ; dans ce cas, on renvoie `false` en séquence.

```
# exception Trouvé;;
L'exception Trouvé est définie.
# let caractère_dans_chaîne chaîne car =
      try
        for i = 0 to string_length chaîne - 1 do
          if chaîne.[i] = car then raise Trouvé
        done;
        false
```

```
      with Trouvé -> true;;
caractère_dans_chaîne : string -> char -> bool = <fun>
```

Ici le déclenchement de l'exception n'est pas un cas d'erreur, mais plutôt un événement heureux : on a détecté la présence du caractère dans la chaîne. On ne peut pas dire non plus que ce soit vraiment un événement exceptionnel, une « exception » au calcul normal : c'est un signal attendu, tout simplement.

Sans le mécanisme des exceptions la fonction précédente devrait être écrite avec une référence initialisée à `false` en début de boucle et mise à `true` lorsqu'on rencontre le caractère.

```
# let car_dans_chaîne chaîne car =
      let trouvé = ref false in
      for i = 0 to string_length chaîne - 1 do
       if chaîne.[i] = car then trouvé := true
      done;
      !trouvé;;
car_dans_chaîne : string -> char -> bool = <fun>
```

Cette version est un peu moins efficace, puisque le parcours de la chaîne est toujours effectué complètement, alors qu'il est inutile de le continuer dès qu'on a détecté la présence du caractère. Cet argument d'efficacité est minime : le choix entre les deux versions est essentiellement une affaire de goût personnel, de style et d'expérience. Nous préférons la version avec exception, car elle se généralise plus facilement à plusieurs événements attendus dans la boucle. Au surplus, la boucle s'arrête instantanément quand l'événement arrive et c'est très souvent un comportement algorithmiquement nécessaire du programme.

7.3 Fonctions de recherche dans les listes

Les réponses toutes faites de Camélia sont stockées dans des listes d'une forme particulière, les listes d'association, qui associent des réponses à certains mots de la phrase du patient.

Appartenance d'un élément à une liste

Nous commençons par écrire la fonction `membre`, qui détermine si son premier argument est élément d'une liste donnée en second argument. Travaillant sur des listes, la fonction `membre` doit par conséquent envisager les deux cas possibles de listes ; d'où le squelette de fonction :

```
let membre e = function
  | [] -> ...
  | x :: reste -> ... ;;
```

Le cas de la liste vide est simple : l'élément à rechercher n'apparaît certainement pas dans la liste.

```
let membre e = function
  | [] -> false
  | x :: reste -> ... ;;
```

Dans le cas général, il faut tester si `e` est égal à la tête de la liste (`x`), sinon interroger récursivement le reste de la liste. On obtient donc :

```
# let rec membre elem = function
    | [] -> false
    | x :: reste -> x = elem || membre elem reste;;
membre : 'a -> 'a list -> bool = <fun>
```

Listes d'association

Les listes d'association sont simplement des listes de paires où le premier élément de la paire est une clé et le second la valeur associée à la clé. (Pour les tenants de Bourbaki, c'est le graphe d'une fonction donnée en extension.) Par exemple, considérez une liste d'adresses. On associe au nom d'une personne son adresse :

```
# let adresses =
      [("Pierre Caml","Domaine de Voluceau, 78153 Le Chesnay Cedex");
       ("Xavier Light","45 rue d'Ulm, 75005 Paris")];;
adresses : (string * string) list =
 ["Pierre Caml", "Domaine de Voluceau, 78153 Le Chesnay Cedex";
  "Xavier Light", "45 rue d'Ulm, 75005 Paris"]
```

Dans notre programme, une liste d'association donnera les réponses possibles associées aux phrases très simples. Voici un extrait de la liste d'association des phrases simples :

```
let réponses_aux_phrases_simples =
   [([],
       [|"Voulez-vous changer de sujet?"; "Continuez"|]);
    (["et"; "alors"],
       [|"Alors expliquez-moi"; "Ne soyez pas si agressif"|]);
    (["non"],
       [|"C'est vite dit"; "Pourriez-vous préciser ?"|]);
    (["oui"],
       [|"C'est un peu rapide"; "Donnez-moi plus de détails"|])];;
```

Cette liste associe à la liste vide (correspondant à une réponse vide) le tableau de réponses possibles `[|"Voulez-vous changer de sujet?"; "Continuez"|]`. Elle associe à la phrase réduite à `et alors?` les réponses `[|"Alors expliquez-moi"; "Ne soyez pas si agressif"|]`. Il nous faut donc retrouver la valeur associée à une phrase dans cette liste d'association. La fonction générale de recherche dans une liste d'association s'écrit très simplement : on teste si la clé a été rencontrée, auquel cas on renvoie la valeur associée ; sinon on cherche dans le reste de la liste. Si la liste est épuisée, il n'y a pas d'espoir de trouver la valeur associée et l'on déclenche l'exception constante `Pas_trouvé`.

```
# exception Pas_trouvé;;
L'exception Pas_trouvé est définie.
# let rec associé_de x = function
    | [] -> raise Pas_trouvé
    | (clé, valeur) :: l ->
        if x = clé then valeur else associé_de x l;;
associé_de : 'a -> ('a * 'b) list -> 'b = <fun>
```

```
# associé_de "Pierre Caml" adresses;;
- : string = "Domaine de Voluceau, 78153 Le Chesnay Cedex"

# associé_de "Xavier Light" adresses;;
- : string = "45 rue d'Ulm, 75005 Paris"

# associé_de "Gérard Coq" adresses;;
Exception non rattrapée: Pas_trouvé
```

Cette fonction est prédéfinie en Caml sous le nom de `assoc`. En cas d'échec, elle déclenche toujours une exception. Dans le système Caml Light c'est l'exception `Not_found` (signifiant « pas trouvé » en anglais). C'est souvent l'exception déclenchée par les fonctions de recherche de la bibliothèque Caml Light.

Des listes d'association multi-clés

Pour les besoins de notre programme nous devons gérer des listes d'association plus compliquées que des listes (clé, valeur). En effet, nous considérons que certains mots « intéressants » sont synonymes et donc que leurs réponses associées sont les mêmes. Par exemple, `ordinateur` et `machine` appellent les mêmes réponses. Le pluriel d'un mot est souvent traité comme le mot lui-même, par exemple `ami` et `amis`. Dans ces listes, une valeur n'est donc plus associée à une seule clé, mais à une liste de clés équivalentes. Nous écrivons donc la fonction qui teste si une clé donnée fait partie de la liste de clés et retourne en ce cas la valeur associée :

```
# let rec associé_dans_liste clé = function
    | [] -> raise Pas_trouvé
    | (liste_de_clés, valeur) :: reste ->
        if membre clé liste_de_clés then valeur
        else associé_dans_liste clé reste;;
associé_dans_liste : 'a -> ('a list * 'b) list -> 'b = <fun>
```

De la même manière, nous avons besoin de chercher, parmi une liste de clés la valeur associée à la première clé qui figure dans les clés d'une liste d'association (listes de clés, valeur). Nous parcourons donc la liste de clés argument en cherchant l'associé de la première clé rencontrée. S'il n'y a pas d'associé à cette clé, nous cherchons, parmi le reste de la liste de clés argument, la première clé qui ait un associé dans la liste d'association. Remarquez au passage que le caractère `'` est autorisé dans les noms d'identificateur en Caml.

```
# let rec associé_d'un_élément_de liste_de_clés liste_d'association =
      match liste_de_clés with
      | [] -> raise Pas_trouvé
      | clé :: reste ->
         try
           associé_dans_liste clé liste_d'association
         with Pas_trouvé ->
           associé_d'un_élément_de reste liste_d'association;;
associé_d'un_élément_de : 'a list -> ('a list * 'b) list -> 'b = <fun>
```

7.4 Traitements de chaînes de caractères

Notre programme va effectuer une certaine normalisation de l'entrée de l'utilisateur : passer systématiquement toute la phrase en minuscules et ôter les signes de ponctuation et accents éventuels, par exemple. Les fonctions correspondantes illustrent le traitement de chaînes de caractères en Caml.

Passage en minuscules

En machine, les caractères sont évidemment enregistrés comme des nombres. Le codage utilisé en Caml s'appelle le code ASCII. Il suffit de le faire imprimer par Caml pour comprendre comment sont rangés les caractères de l'alphabet (remarquez aussi les chiffres) :

```
# for i = 32 to 126 do
    if i < 100 then print_string " ";
    print_int i; print_string " ";
    print_char (char_of_int i); print_string "  ";
    if i mod 8 = 7 then print_newline ()
  done;
  print_newline ();;
 32     33 !   34 "   35 #   36 $   37 %   38 &   39 '
 40 (   41 )   42 *   43 +   44 ,   45 -   46 .   47 /
 48 0   49 1   50 2   51 3   52 4   53 5   54 6   55 7
 56 8   57 9   58 :   59 ;   60 <   61 =   62 >   63 ?
 64 @   65 A   66 B   67 C   68 D   69 E   70 F   71 G
 72 H   73 I   74 J   75 K   76 L   77 M   78 N   79 O
 80 P   81 Q   82 R   83 S   84 T   85 U   86 V   87 W
 88 X   89 Y   90 Z   91 [   92 \   93 ]   94 ^   95 _
 96 `   97 a   98 b   99 c  100 d  101 e  102 f  103 g
104 h  105 i  106 j  107 k  108 l  109 m  110 n  111 o
112 p  113 q  114 r  115 s  116 t  117 u  118 v  119 w
120 x  121 y  122 z  123 {  124 |  125 }  126 ~
- : unit = ()
```

Le passage en minuscule revient donc à un simple calcul sur le code ASCII du caractère : si le caractère est une majuscule, on lui ajoute 32 pour obtenir la minuscule correspondante.

```
# let minuscule_de car =
      if int_of_char car >= 65 && int_of_char car <= 90
      then char_of_int (int_of_char car + 32)
      else car;;
minuscule_de : char -> char = <fun>
```

Pour passer une chaîne de caractères tout entière en minuscules, il suffit d'itérer la fonction précédente.

```
# let minuscules chaîne =
    let chaîne_en_minuscules = create_string (string_length chaîne) in
    for i = 0 to string_length chaîne - 1 do
      chaîne_en_minuscules.[i] <- minuscule_de chaîne.[i]
    done;
```

```
    chaîne_en_minuscules;;
minuscules : string -> string = <fun>
```

Nous avons également besoin d'extraire une sous-chaîne d'une chaîne. La sous-chaîne est repérée par ses indices de début et de fin. On utilise la fonction prédéfinie `sub_string` qui calcule une sous-chaîne partant d'un indice donné et d'une longueur donnée :

```
# sub_string "Caml" 0 3;;
- : string = "Cam"

# let sous_chaîne s départ fin =
      sub_string s départ (fin - départ + 1);;
sous_chaîne : string -> int -> int -> string = <fun>
```

Suppression des accents

La simplification d'un mot consiste à supprimer les accents, que l'on considère comme non significatifs. On admet également que l'utilisateur ait pu taper `e^` pour `ê` ou `e'` pour `é`. Enfin, on supprime les articles élidés qui apparaissent éventuellement au début du mot. Par exemple, `l'air` devient `air`.

Plutôt que de « mettre en dur » ces conventions dans le code de la fonction de simplification, nous écrivons une fonction générale, paramétrée par une table de simplifications, qui n'est autre qu'une liste d'association entre chaînes.

```
# let simplifications =
    [("à","a"); ("ç","c"); ("é","e"); ("è","e"); ("ê","e"); ("ù","u");
     ("a‘","a"); ("e’","e"); ("e‘", "e"); ("e^","e"); ("u‘","u");
     ("qu’", ""); ("l’", ""); ("d’", "")];;
```

La fonction de simplification consiste à recopier le mot argument dans une nouvelle chaîne. Les caractères sont recopiés un par un, sauf si le caractère courant et les caractères suivants forment une des chaînes à simplifier, auquel cas on les remplace par la chaîne associée dans la table de simplifications.

```
# let simplifie_mot mot =
      let nouveau_mot = create_string (string_length mot) in
      let i = ref 0 and j = ref 0 in
      let rec cherche_traduction = function
      | [] -> raise Pas_trouvé
      | (original, traduction) :: reste ->
          let longueur = string_length original in
          if !i + longueur <= string_length mot
          && sub_string mot !i longueur = original
          then (longueur, traduction)
          else cherche_traduction reste in
      while !i < string_length mot do
        try
          let (longueur, traduction) =
            cherche_traduction simplifications in
          blit_string traduction 0 nouveau_mot !j
                      (string_length traduction);
          i := !i + longueur;
```

```
          j := !j + string_length traduction
        with Pas_trouvé ->
          nouveau_mot.[!j] <- mot.[!i];
          i := !i + 1;
          j := !j + 1
      done;
      sub_string nouveau_mot 0 !j;;
simplifie_mot : string -> string = <fun>
```

Division en mots

La division d'une chaîne de caractères en mots est une autre opération délicate. Elle consiste à parcourir la chaîne (à l'envers) à la recherche des séparateurs. Dès qu'un séparateur est trouvé, on extrait un mot qu'on ajoute à la liste de mots `mots`. On maintient un compteur `j` qui indique le dernier caractère du mot courant, tandis que le compteur de boucle `i` sert à en repérer le début. Notez que le caractère « fin de ligne » est écrit `\n`. À l'intérieur d'une chaîne de caractères, la notation `\n` représente aussi un retour à la ligne.

```
# let divise_en_mots chaîne =
      let mots = ref [] in
      let j = ref (string_length chaîne - 1) in
      let ajoute_mot i j =
          if i <= j then
          mots := simplifie_mot (sous_chaîne chaîne i j) :: !mots in
      for i =  string_length chaîne - 1 downto 0 do
        match chaîne.[i] with
        | (‘ ‘ | ‘\n‘ | ‘.‘ | ‘,‘ | ‘;‘ | ‘-‘ | ‘!‘ | ‘?‘) ->
           ajoute_mot (i + 1) !j; j := i - 1
        | _ -> ()
     done;
     ajoute_mot 0 !j;   (* extraction du dernier mot *)
     !mots;;
divise_en_mots : string -> string list = <fun>
```

Nous rencontrons un trait nouveau du langage : les barres verticales à l'intérieur des filtres, comme dans `‘ ‘ | ‘\n‘ | ...` Ces filtres sont des filtres à plusieurs cas que l'on appelle les filtres « ou ». Ils filtrent l'union des cas filtrés par leurs composants. C'est une simple facilité syntaxique, qui évite de recopier plusieurs fois le corps de la clause. Par exemple, la clause `(1 | 2) -> true` est équivalente aux deux clauses `1 -> true | 2 -> true`.

7.5 Camélia

La base de données

Le principe de `Camélia` est donc d'utiliser un ensemble de phrases écrites pour elle par le programmeur. Le bon choix de ces phrases intervient beaucoup dans la qualité de l'échange. Nous donnons donc ici ce que nous appelons pompeusement la « base de

données » de Camélia, qui n'est autre qu'un ensemble de listes d'association simples ou multi-clés. La base de données comprend les listes suivantes :

- `salutations` : le mot de la fin de Camélia, typiquement « Ce sera long et difficile, revenez me voir souvent ... ».
- `relances` : les phrases utilisées quand le programme ne sait plus quoi dire, typiquement « Parlez-moi encore de vous » ou « Êtes-vous marié ? ».
- `réponses_types` : ce que le programme répond quand il détecte un caractère « ? » et rien de plus intéressant dans la phrase. Typiquement « C'est moi qui pose les questions ».
- `réponses_aux_phrases_simples` : pour répondre aux phrases à l'emporte-pièce comme « Et alors ? » ou « Oui ». Réponse typique : « Alors, expliquez-moi » ou « Donnez-moi plus de détails ».
- `réponses_aux_petits_mots` : quand le programme n'a rien de mieux à faire, il cherche un mot très souvent employé en français qui lui permette de faire une réponse sensée, sans rien comprendre. Exemple typique : si la phrase contient le mot « jamais » le programme peut répondre « Jamais me semble un peu fort, non ? ».
- `réponses_aux_mots_intéressants` : le programme surveille si le patient emploie des mots « psychanalytiques », comme père ou famille, et réagit alors en conséquence. Réponses typiques : « Racontez-moi vos problèmes » quand le mot « malheureux » est détecté.

```
let salutations =
[|"Ce sera long et difficile, revenez me voir \
   souvent ...";
  "Votre cas n'est pas simple, et même assez \
   inquiétant ... A bientôt?";
  "Diagnostic simple: sans conteste vous êtes \
   paranoïaque.";
  "Avec une probabilité de 92.37234%: \
   perversion polymorphe.";
  "Vous souffrez d'une schizophrénie en rapide \
   évolution, DANGER";
  "D'après mes calculs, votre santé mentale est \
   compromise.";
  "Mon ultime conseil: il ne faut pas rester \
   comme cela, soignez-vous!"|];;
let relances =
[| "Parlez-moi un peu de vous";
   "Êtes-vous marié?";
   "Avez-vous des enfants?";
   "Parlons de votre entourage";
   "Aimez-vous la vie?";
   "Aimez-vous ce moyen de communiquer?";
   "Parlons de votre famille";
   "Parlez-moi encore de vous";
   "Que pensez-vous des ordinateurs?";
   "Que pensez-vous de Linux?";
   "Que pensez-vous de Caml?";
   "De quoi parlerons-nous maintenant?";
   "Avez-vous beaucoup d'amis?";
   "Avez-vous de graves problèmes?";
   "Parlez-moi de vos problèmes";
   "Faites-vous des rêves étranges?";
   "Faites-vous souvent des cauchemars?";
   "Que pensez-vous de l'amour?";
   "Que pensez-vous de la sexualité?";
   "Quels sont vos violons d'Ingres?";
   "Qu'est-ce qui vous intéresse dans la vie?";
   "Que pensez-vous de la vie en général?"|];;
let réponses_types =
[| "C'est moi qui pose les questions";
   "Je ne suis pas là pour répondre à vos \
    questions";
   "Question très intéressante, mais qu'en \
    pensez-vous?";
   "Quelle question!";
   "Pourquoi me posez-vous cette question?";
   "Vous le savez très bien";
   "La réponse est sans importance";
   "Vous le dire ne vous apporterait rien";
   "Un psychanalyste n'a pas le droit de \
    répondre à ce genre de questions";
   "Je n'ai pas le droit de vous répondre";
   "Il m'est interdit de vous le dire";
   "Vous ne comprendriez pas";
   "Permettez-moi de ne pas répondre";
   "Laissez-moi réfléchir. Pouvez-vous \
    reformuler la question?";
   "Je ne suis pas certaine de bien comprendre \
    la question";
   "Je ne sais pas";
   "Cherchez un peu";
   "C'est évident pour tout le monde, sauf pour \
```

```
    vous; réfléchissez!";
   "C'est à vous de trouver la réponse";
   "Cherchez bien au fond de vous-même, vous le \
    savez en fait"|];;
let réponses_aux_phrases_simples =
[([],
  [|"Voulez-vous changer de sujet?";
    "Continuez";
    "Continuez, vous m'intéressez";
    "Je vous écoute";
    "Vous n'avez rien d'autre à dire?";
    "Continuez, je vous prie";
    "C'est tout ce que vous avez à dire?";
    "M'avez-vous tout dit là-dessus?";
    "Je n'en sais pas encore assez sur vous; \
     continuez"|]);
 (["quoi"],
  [|"Excusez-moi je pensais à autre chose, \
     continuons";
    "Réfléchissez";
    "Changeons de sujet, s'il vous plaît";
    "Je me comprends";
    "Il me semblait pourtant avoir été claire";
    "La communication est difficile, non?";
    "Ah les hommes! Ils ne comprennent rien!";
    "Cessez de poser des questions";
    "N'auriez-vous pas des problèmes à me \
     comprendre?"|]);
 (["non"],
  [|"C'est vite dit";
    "Pourriez-vous préciser?";
    "Je note: c'est non";
    "Mais encore?";
    "La réponse n'est pas si simple, non?";
    "Vous êtes vraiment très sûr de vous";
    "Ne vous arrive-t-il pas de douter de \
     vous-même?";
    "Ne répondez pas toujours oui ou non";
    "Syndrôme du yes/no. Expliquez-vous, que \
     diable!";
    "Au moins vous ne souffrez pas de diarrhée \
     verbale";
    "Comment pouvez-vous être si sûr de \
     vous?"|]);
 (["si"],
  [|"Si bémol?";
    "D'accord, d'accord";
    "Mouais, je m'en doutais un peu, \
     figurez-vous";
    "Expliquez-vous, ``si'' ne me suffit pas";
    "Réponse trop laconique";
    "Syndrôme du si";
    "Vous n'êtes pas bavard vous au moins"|]);
 (["oui"],
  [|"C'est un peu rapide";
    "Donnez-moi plus de détails";
    "Vous pourriez préciser?";
    "Je voudrais comprendre pourquoi";
    "La réponse n'est pas si simple, non?";
    "C'est franc et massif au moins";
    "Ça ne m'en dit pas vraiment plus, \
     expliquez-moi pourquoi.";
    "Vous êtes sûr?";
    "Soyez moins bref: développez";
    "Plus laconique tu meurs";
    "Si vous ne m'expliquez pas mieux, comment \
     vous comprendre?";
    "Ne répondez pas toujours oui ou non";
    "Dont acte";
    "Et pour quelles raisons?"|]);
 (["et"; "alors"],
  [|"Alors, expliquez-moi";
    "Ne soyez pas si agressif";
    "Alors j'aimerais avoir plus d'informations \
     là-dessus";
    "Zorro est arrivé";
    "Et alors, et alors, expliquez-vous!";
    "C'était un test pour savoir si vous \
     suiviez"|]);
 (["encore"],
  [|"On peut changer de sujet, si vous voulez?";
    "Il faut bien crever l'abcès!";
    "Les choses importantes doivent être \
     dites!";
    "Il faut savoir affronter les problèmes";
    "Je suis plus têtue que vous!";
    "Pensez-vous que je radote?";
    "Dites tout de suite que je suis \
     gâteuse!"|])
];;
let réponses_aux_petits_mots =
[(["nest"],
  [|"Pas du tout?";
    "Vraiment pas?";
    "Pourquoi pas?"|]);
 (["jamais"],
  [|"Ne dites jamais ``jamais''";
    "Jamais me semble un peu fort, non?";
    "Jamais?"|]);
 (["non"],
  [|"En êtes vous sûr?";
    "Pourquoi pas?";
    "Que diriez-vous dans le cas contraire?";
    "C'est une opinion défendable";
    "Je saurai au moins votre opinion \
     là-dessus"|]);
 (["rien"],
  [|"Rien du tout?";
    "Pourquoi pas?";
    "Que diriez-vous dans le cas contraire?";
    "Voilà qui est franc";
    "Au moins c'est clair";
    "Même pas un petit peu?";
    "Rien est un peu exagéré, non?"|]);
 (["pourquoi"],
  [| "Parce que";
     "Je ne réponds pas aux questions des \
      malades";
     "Si vous le savez pas, ce n'est pas à moi \
      de vous l'apprendre";
     "Personne ne peut répondre à cette \
      question";
     "Pensez-vous qu'une machine peut répondre \
      à ça?";
     "Ce serait trop long à expliquer";
     "Je sais bien pourquoi, mais vous ne \
      comprendriez pas";
     "C'est difficile à dire"|]);
 (["aucun"],
  [|"Vraiment aucun?";
```

```
    "Pas le moindre?";
    "Le regrettez-vous?";
    "C'est un fait nouveau pour moi"|]);
  (["pas"],
   [|"Ça me semble un peu négatif";
    "Vraiment?";
    "Pourquoi cela?";
    "Je ne m'en serais pas doutée";
    "Difficile";
    "J'ai l'habitude d'entendre ça";
    "Êtes vous troublé à ce point?";
    "Vous ne devriez pas parler ainsi"|]);
  (["sait"; "sais"; "savoir"],
   [|"Le savoir est une denrée rare";
    "Êtes-vous certain de le savoir?";
    "Ne subsiste-t-il pas de doute?";
    "Je ne pourrais pas en dire autant";
    "Difficile à admettre";
    "En êtes-vous si sûr?"|]);
  (["oui"],
   [|"En êtes-vous certain?";
    "Vous êtes vraiment sûr de vous";
    "Ça ne me semblait pas évident";
    "Pourtant j'aurais cru le contraire";
    "C'est intéressant, continuez";
    "Quelle affirmation sans détours";
    "Très bien";
    "Quel aveu!";
    "Bon"|]);
  (["quoi";"comment"],
   [|"C'est à vous de me le dire";
    "Difficile à dire";
    "Réfléchissez, vous comprendrez";
    "La réponse est en vous"|]);
  (["merci";"remercie"],
   [|"Ne me remerciez pas";
    "Je suis là pour vous aider";
    "Allez allez, continuez";
    "C'est tout naturel";
    "C'était vraiment facile"|])
];;
let réponses_aux_mots_intéressants =
[(["peur";"peurs"],
  [|"Parlez-moi de vos frayeurs";
    "Avez-vous souvent peur?";
    "Avez-vous des peurs inexpliquées, des \
     cauchemars?"|]);
 (["mort"; "morte"; "morts"],
  [|"Je vous plains beaucoup";
    "La mort est un sujet très grave";
    "Il faut essayer de prendre le dessus";
    "Il faut pourtant quelquefois affronter la \
     mort";
    "C'est malheureux";
    "Essayez de ne plus y penser"|]);
 (["malheureux"; "malheureuse";
   "probleme"; "problemes"],
  [|"Racontez-moi vos problèmes";
    "Quels malheurs sont les vôtres?";
    "Avez-vous vraiment des raisons de vous \
     plaindre?";
    "Le bonheur existe aussi vous savez."|]);
 (["malheur"; "malheurs"],
  [|"Malheur est peut-être exagéré, non?";
    "Le malheur est une notion relative. \
     Qu'entendez-vous par malheur?";
    "Bonheur, malheur, je n'entends parler que \
     de ça. Continuez."|]);
 (["ennui"; "ennuies"; "ennuyez"],
  [|"L'ennui, ça dépend de vous";
    "Est-ce que je vous ennuie?";
    "Je le regrette pour vous";
    "C'est dommage pour vous"|]);
 (["ennuis"],
  [|"Les ennuis sont souvent passagers";
    "Tout ne peut pas être rose, n'est-ce pas?";
    "Quelle tristesse, n'est-ce pas?";
    "Est-ce vraiment très grave?"|]);
 (["ordinatuer"],
  [| "Vous voulez dire ordinateur, je \
      suppose"|]);
 (["ordinateur"; "ordinateurs"; "machine"; \
    "machines"],
  [|"Connaissez-vous bien l'informatique?";
    "Changeons de sujet, celui-là ne \
     m'intéresse pas";
    "Ah les machines!";
    "Les machines c'est si bête!";
    "Je connais bien les ordinateurs, et \
     j'évite de les fréquenter!";
    "Je n'ai pas d'avis sur les machines en \
     général";
    "Vous savez, je suis une machine moi-même \
     ..."|]);
 (["informatique"; "informaticien"; \
    "informaticiens"],
  [|"Quel beau métier de s'occuper des \
     machines";
    "Ah l'informatique!";
    "L'informatique est un dur métier";
    "C'est difficile l'informatique, non?";
    "Aimez-vous vraiment l'informatique?";
    "Vous n'aimez pas follement l'informatique, \
     m'a-t'on dit"|]);
 (["famille"],
  [|"Avez-vous des frères et soeurs?";
    "Parlez-moi de votre père";
    "Parlez-moi de votre mère";
    "Voilà qui m'intéresse énormément";
    "Dites-m'en plus sur votre famille";
    "La famille c'est souvent compliqué"|]);
 (["pere"],
  [|"Ressemblez-vous à votre père?";
    "Parlez-moi encore de votre père";
    "Et votre mère?";
    "Votre père?"|]);
 (["mere"],
  [|"Ressemblez-vous à votre mère ou à votre \
     père?";
    "Parlez-moi de votre mère";
    "Parlez-moi encore de votre mère";
    "Et votre père?";
    "Votre mère?"|]);
 (["ami"; "amis"; "amie"; "amies"; "copains"; \
    "copines"],
  [|"Avez-vous beaucoup d'amis?";
    "Comment vous êtes-vous connus?";
    "Comment cela se passe-t-il avec vos amis?";
    "Avez-vous de fréquentes disputes avec vos \
     amis?";
```

```
   "Des amies?";
   "Des petites amies?";
   "Des petits amis?";
   "Depuis combien de temps vous \
    connaissez-vous?"|]);
 (["deteste"; "hais"],
  [|"Est-ce raisonnable de détester à ce \
     point?";
    "Le mot n'est-il pas un peu fort?";
    "Modérez un peu vos sentiments"|]);
 (["mari"],
  [|"Êtes-vous depuis longtemps ensemble?";
    "Comment l'avez-vous rencontré?";
    "Pensez-vous qu'il faille être fidèle à son \
     mari?"|]);
 (["amour"],
  [|"Et l'amour fou, qu'en pensez-vous?";
    "C'est compliqué l'amour, non?";
    "L'amour, l'amour, le connaissez-vous \
     vraiment?";
    "Avez-vous déjà connu l'amour?";
    "Connaissez-vous le grand amour?";
    "L'amour, comment l'avez-vous \
     rencontré?"|]);
 (["argent"],
  [|"Faute d'argent, c'est douleur sans \
     pareille";
    "Avez-vous des problèmes d'argent?";
    "L'argent a beaucoup de connotations, \
     continuez sur le sujet";
    "Aimez-vous beaucoup l'argent?";
    "Avez-vous peur de manquer d'argent?"|]);
 (["caml"],
  [|"Vous voulez dire les cigarettes Camel?";
    "J'ai entendu parler de ce remarquable \
     langage Caml";
    "Tout ce que vous allez dire pourra être \
     retenu contre vous";
    "Sans Caml je ne serais pas là; je refuse \
     donc d'en parler";
    "A mon avis, Caml est sans égal";
    "Ce langage Caml est clairement en avance \
     sur nombre de ses successeurs!";
    "Caml est puissant, et quelle belle \
     syntaxe, hein?";
    "Caml, c'est vraiment facile";
    "Caml, c'est un langage de la sixième \
     génération, non?";
    "C'est vrai que si Caml n'existait pas, il \
     faudrait l'inventer d'urgence!";
    "Je suis catégorique: Caml est un langage \
     très simple!";
    "En Caml, c'est trop simple: les programmes \
     marchent toujours!";
    "Un tel langage, quelle aubaine pour les \
     humains!";
    "Caml, ça c'est du langage!"
  |]
 );
 (["sml"],
  [|"Pas de provocation s'il vous plaît";
    "Ne me parlez pas des mammouths";
    "SML, dites-vous?";
    "Jamais entendu parler de SML, c'est \
     quoi?";
    "Faudrait savoir est-ce ML ou pas?"|]);
 (["langage"; "langages"],
  [|"Vous voulez dire langage de \
     programmation?";
    "Je ne connais que le langage Caml";
    "Connaissez-vous bien le langage Caml?";
    "Hors de Caml, point de salut, non?";
    "A mon avis, Caml est sans égal";
    "Oui, c'est puissant, mais quelle syntaxe!";
    "Et les problèmes de syntaxe?"
  |]
 );
 (["programme"; "programmes"],
  [|"Vous parlez de programmes d'ordinateur?";
    "Il y a souvent des erreurs dans vos \
     programmes, non?";
    "Connaissez-vous vraiment la \
     programmation?";
    "Vos programmes s'écriraient plus \
     naturellement en Caml";
    "Vos programmes s'écriraient plus \
     simplement en Caml";
    "A mon avis, la programmation c'est facile, \
     non?";
    "Avez-vous des problèmes avec vos \
     programmes?"
  |]
 );
 (["chameaux"; "chameau"],
  [|"Le chameau est un charmant animal d'une \
     grande sobriété, non?";
    "Le chameau est mon animal favori, pas \
     vous?";
    "Certes le chameau est d'un caractère un \
     peu difficile, mais il en est de \
     charmants, n'est-ce-pas?";
    "Un chameau à deux bosses ou un \
     dromadaire?";
    "Qu'avez-vous de plus à dire sur les \
     chameaux?"|]);
 (["naime"],
  [|"Même pas un peu?";
    "Détestez-vous carrément?";
    "Pourquoi cette répulsion?";
    "Aimer me semble un sentiment étrange, pas \
     vous?";
    "Peut-on aimer vraiment?";
    "Aimer ne pas aimer est-ce vraiment la \
     question?"|]);
 (["aime"],
  [|"Beaucoup?";
    "Sans aucune retenue?";
    "Pourquoi cette attirance?";
    "Comment expliquer ce sentiment?";
    "Peut-on aimer vraiment?";
    "Aimer ne pas aimer est-ce vraiment la \
     question?"|]);
 (["sexe"],
  [|"Personnellement je ne suis pas concernée";
    "Ça paraît intéressant!";
    "On m'a dit que le sexe est important pour \
     les humains";
    "Le sexe d'accord, mais l'amour?";
    "Avez-vous entendu parler du Sida?"|]);
 (["cauchemar"; "cauchemars"; "reve"; "reves"],
```

```
  [|"J'ai du mal à comprendre; je ne rêve \
     jamais!";
    "Vos activités nocturnes m'intéressent. \
     Continuez";
    "Ça me paraît bizarre!";
    "Les cauchemars vous réveillent-ils la \
     nuit?";
    "Avez-vous des insomnies?";
    "Faites-vous beaucoup de cauchemars?";
    "Faites-vous souvent des rêves étranges?";
    "Que pensez-vous de l'hypnose?"|]);
 (["anxieux"; "anxieuse"],
  [|"L'anxiété est une vraie maladie";
    "Les anxieux ont souvent des problèmes avec \
     leur entourage. L'avez-vous remarqué?";
    "L'anxiété est une vraie souffrance, \
     non?"|]);
 (["stupide"; "idiot"],
  [|"Pensez-vous que ce soit un crime d'être \
     stupide?";
    "J'ai d'excellents amis qui sont stupides \
     aussi";
    "La sottise est la chose du monde la mieux \
     partagée";
    "Ne soyez pas stupide non plus";
    "Vous-même, n'êtes-vous pas stupide \
     quelquefois?";
    "Ne pensez-vous pas que c'est quelquefois \
     utile d'être stupide?"|]);
 (["femme"],
  [|"Êtes-vous depuis longtemps ensemble?";
    "Comment votre rencontre s'est-elle \
     passée?";
    "Aimez-vous cette femme?";
    "Est-ce une femme ordinaire?"|]);
 (["mal"; "difficile"],
  [|"Je vous plains beaucoup";
    "Êtes-vous certain d'être objectif?";
    "Je peux tenter de vous aider";
    "Et c'est tout ce que vous vouliez me \
     dire?";
    "Est-ce pour cela que vous vous êtes \
     adressé à moi?"|]);
 (["fatigue"],
  [|"La fatigue n'est pas une maladie";
    "Quand on est fatigué ne faut-il pas se \
     reposer?";
    "Je suis une machine: je ne connais pas la \
     fatigue";
    "Ah frêles humains qui connaissez la \
     fatigue";
    "Que pensez-vous de la fatigue en général?";
    "Pourquoi pensez-vous que ça vaut la peine \
     de se fatiguer?";
    "Les gens fatigués le sont souvent de leur \
     fait, non?"|]);
 (["tu"; "vous"; "toi"],
  [|"Ne parlons pas de moi";
    "Parlons de vous, c'est plus important";
    "Si on parlait de vous?";
    "Moi, je ne suis qu'une machine ...";
    "Moi?";
    "Excusez-moi";
    "Ne m'en veuillez pas si je vous interroge. \
     Continuez";
    "Vous ne le pensez pas vraiment?"|])
];;
```

7.6 Dialogue avec l'utilisateur

Tirage aléatoire

Pour donner un peu de variété au dialogue, il nous faut évidemment une fonction qui choisisse un élément au hasard dans un tableau de valeurs possibles. C'est très simple en utilisant le générateur aléatoire de nombres entiers fourni par Caml. Dans le système Caml Light, il s'agit de la fonction `random__int`, qui renvoie un entier compris entre 0 (inclus) et son argument entier (exclu). Il suffit donc de l'appeler avec la longueur du tableau.

```
# let au_choix_dans v = v.(random__int (vect_length v));;
au_choix_dans : 'a vect -> 'a = <fun>
```

Les utilitaires de salutations

Un utilitaire pour écrire des messages au terminal :

```
# let message s = print_string s; print_newline ();;
message : string -> unit = <fun>
# message (au_choix_dans salutations);;
Diagnostic simple: sans conteste vous êtes paranoïaque.
```

```
- : unit = ()
# message (au_choix_dans salutations);;
Votre cas n'est pas simple, et même assez inquiétant ... A bientôt?
- : unit = ()
```

Deux petites fonctions pour dire bonjour et au revoir, en gérant le prix de la consultation :

```
# let prix_à_payer = ref 0;;
prix_à_payer : int ref = ref 0
# let bonjour () =
      prix_à_payer := 40;
      message
       "\nBonjour, je m'appelle Camélia.\n\nJe suis là \
        pour vous aider à résoudre vos problèmes psychologiques.\
        \nTerminez en me disant: Au revoir.\n\
        \nAllons-y. Parlez-moi de vous.\n";;
bonjour : unit -> unit = <fun>
# let au_revoir () =
      message "\nLe résultat de mes observations:\n";
      message (au_choix_dans salutations);
      message "\nAu revoir ...\n";
      print_string "Vous me devez "; print_int !prix_à_payer;
      message " euros. Chèque à l'ordre de Camélia. Merci.";;
au_revoir : unit -> unit = <fun>
```

Pour lire la réponse du patient, on utilise la fonction prédéfinie `read_line` qui lit une ligne tapée au clavier. De manière très réaliste, chaque échange entre le patient et Camélia accroît la note de la consultation.

```
# let écoute_le_patient () =
      prix_à_payer := !prix_à_payer + 2;
      print_string ">> ";
      read_line ();;
écoute_le_patient : unit -> string = <fun>
```

Pour simplifier le travail de recherche des phrases courtes, on utilise une fonction qui reconnaît les phrases synonymes. Par exemple, la phrase `comment?` est assimilée à `quoi?`. De même, `bien sûr`, `oui` et `bien sûr que oui` sont assimilées à `oui`.

```
# let rec synonyme_de_phrase = function
    | ["comment"] -> ["quoi"]
    | ["bien";"sur"] -> ["oui"]
    | "bien"::"sur"::"que"::suite -> synonyme_de_phrase suite
    | (["evidemment"] | ["certainement"]) -> ["oui"]
    | "pas"::"du"::"tout"::_ -> ["non"]
    | phrase -> phrase;;
synonyme_de_phrase : string list -> string list = <fun>
```

La boucle de dialogue

Nous en arrivons maintenant au cœur du programme : les fonctions qui « interprètent » les réponses de l'utilisateur. Commençons par un petit utilitaire pour déterminer si la

consultation est terminée. C'est le cas si la phrase tapée est « Au revoir » ou « Salut ». Rappelons que le caractère `'` est autorisé dans les identificateurs (ainsi `x'` et `x''` sont des noms utilisables en Caml) : on peut donc choisir tout naturellement `c'est_fini`.

```
# let c'est_fini ph = (ph = ["au"; "revoir"]) || (ph = ["salut"]);;
c'est_fini : string list -> bool = <fun>
```

Pour répondre au patient, la fonction `répond_au_patient` se contente d'essayer successivement ses stratégies prédéfinies et d'écrire au terminal la réponse trouvée. Pour cela, on commence par passer le texte du patient en minuscules, puis on le transforme en une liste de mots, qu'on appelle `phrase`. Si cette phrase indique la fin de la consultation, on lance l'exception `Fini`. Sinon, on cherche l'ensemble des réponses possibles pour la phrase donnée en essayant de trouver un associé à la phrase ou à l'un de ses mots, dans la base de données de Camélia. Lorsque l'une de ces tentatives échoue, elle déclenche forcément l'exception `Pas_trouvé`, qui est rattrapée pour essayer la stratégie suivante. Finalement, la stratégie par défaut est de choisir une phrase de relance de la conversation ; c'est donc la clause `with` du dernier `try`. Lorsque les réponses possibles à la phrase entrée sont trouvées, il ne reste qu'à en choisir une au hasard et à l'imprimer.

```
# exception Fini;;
L'exception Fini est définie.
# let répond_au_patient réponse =
      let r = minuscules réponse in
      let phrase = divise_en_mots r in
      if c'est_fini phrase then raise Fini else
      let réponses_possibles =
          try associé_de (synonyme_de_phrase phrase)
                         réponses_aux_phrases_simples
          with Pas_trouvé ->
          try associé_d'un_élément_de phrase
              réponses_aux_mots_intéressants
          with Pas_trouvé ->
          if caractère_dans_chaîne r `?`
          then réponses_types
          else try associé_d'un_élément_de phrase
                   réponses_aux_petits_mots
               with Pas_trouvé -> relances in
      message (au_choix_dans réponses_possibles);
      print_newline ();;
répond_au_patient : string -> unit = <fun>
```

Le programme lui-même itère les fonctions `écoute_le_patient` et `répond_au_patient` en surveillant la fin de l'interaction. L'interruption inopinée de la consultation est rattrapée, pour rappeler la politesse à l'utilisateur (l'interruption d'un programme Caml déclenche une exception prédéfinie, `sys__Break` dans le système Caml Light).

```
# let camélia () =
      bonjour ();
      try
       while true do
         répond_au_patient(écoute_le_patient ());
       done
      with Fini -> au_revoir ()
```

```
        | sys__Break ->
            message "\n\n\nVous pourriez être poli \
                     et me dire au revoir ...\n\n\n";
            au_revoir ();;
camélia : unit -> unit = <fun>
```

7.7 Exemple de session

Tout est prêt pour qu'un des auteurs se fasse soigner :

```
# camélia ();;
Bonjour, je m'appelle Camélia.
Je suis là pour vous aider à résoudre vos problèmes psychologiques.
Terminez en me disant: Au revoir.
Allons-y. Parlez-moi de vous.
>> J'ai des tas de problèmes docteur.
Quels malheurs sont les vôtres?
>> Voilà mon petit chat est mort
La mort est un sujet très grave
>> Oui je ne me console pas
En êtes-vous certain?
>> Oui
Si vous ne m'expliquez pas mieux, comment vous comprendre?
>> Voilà: c'était un chat angora vert qui rêvait beaucoup la nuit
Parlez-moi un peu de vous
>> Ce n'est pas le sujet. C'est mon chat qui m'intéresse
   et l'informatique évidemment.
Aimez-vous vraiment l'informatique?
>> Ça c'est sûr. Surtout quand j'arrive à faire tourner mes
   programmes.
A mon avis, la programmation c'est facile, non?
>> Ben, pas vraiment si vous voulez savoir!
Vous ne le pensez pas vraiment?
>> Si bien su^r.
Que pensez-vous de l'amour?
>> Quoi?
Il me semblait pourtant avoir été claire
>> OK. Revenons à mes problèmes. Je suis malheureux à cause de mon chat.
Racontez-moi vos problèmes
>> Mes enfants ne dorment plus depuis la mort de ce chat
Essayez de ne plus y penser
>> C'est pas facile: ils pleurent tout le temps...
Que pensez-vous de la vie en général?
>> Vaste question.
Que pensez-vous de Caml?
>> Voilà qui est franchement précis. C'est un superbe langage de programmation.
Je ne connais que le langage Caml
>>  Ça m'étonne que vous connaissiez un langage de programmation.
Parlons de vous, c'est plus important
>> Oui. J'aime ma femme, mes enfants, les fins de semaine à la mer
Est-ce une femme ordinaire?
```

```
>> Non, si vous saviez combien elle est extraordinaire
Moi, je ne suis qu'une machine ...
>> Bien sympathique au demeurant
Que pensez-vous de Linux?
>> N'auriez-vous pas un léger tropisme vers l'informatique ?
Excusez-moi
>> Vous êtes toute excusée.
Ne m'en veuillez pas si je vous interroge. Continuez
>> Mon chat angora est mort vous disais-je
Je vous plains beaucoup
>> Oui mais cela ne me console pas vraiment. Que faut-il faire ?
Permettez-moi de ne pas répondre
>> Mais qui pourra m'aider alors ?
Pourquoi me posez-vous cette question ?
>> Parce que j'ai besoin d'aide évidemment!
Parlez-moi encore de vous
>> J'ai besoin d'aide !
Que pensez-vous des ordinateurs ?
>> Ce sont des machines complètement idiotes
Connaissez-vous bien l'informatique ?
>> Un peu! Je suis informaticien professionnel!
Quel beau métier de s'occuper des machines
>> Oui mais les langages, c'est encore plus rigolo!
Hors de Caml, point de salut, non?
>> Mais doit-on ne parler que de Caml ?
Sans Caml je ne serais pas là; je refuse donc d'en parler
>> Mais pourquoi ? Ca m'intéresse beaucoup!
Il m'est interdit de vous le dire
>> Ah bon ? C'est le programmeur qui vous en a empêchée ?
Moi, je ne suis qu'une machine ...
>> Bien sympathique au demeurant. Bon je crois que l'exemple suffira.
   Je vais me coucher. Au revoir Camélia
Qu'est-ce qui vous intéresse dans la vie ?
>> Ca suffit!
Faites-vous souvent des cauchemars?
>> Salut.
Le résultat de mes observations:
Avec une probabilité de 92.37234%: perversion polymorphe.
Au revoir ...
Vous me devez 110 euros. Chèque à l'ordre de Camélia. Merci.
- : unit = ()
```

7.8 Pour aller plus loin

Notre programme est basé sur les idées de Joseph Weizenbaum et de son système Eliza (*Communications of the ACM* n°9, janvier 1966, et n°10, août 1967). Le but de Joseph Weizenbaum était d'explorer la compréhension que pouvait avoir un ordinateur de la conversation d'un humain. Eliza était bien plus sophistiqué que notre Camélia : c'était un programme générique d'analyse de textes et de reconnaissance de mots dans des textes, qu'on pouvait paramétrer par un domaine particulier. Par exemple, pour

le domaine psychologique, on obtenait le programme Doctor, qui existe encore sur de nombreux systèmes Unix. Eliza était capable d'interpréter les mots selon le contexte dans lequel ils apparaissaient et de tenir à jour une représentation globale de ce qui s'était déjà dit entre l'ordinateur et l'utilisateur. Il pouvait ainsi apprendre des faits dans un domaine précis, puis les utiliser ensuite à bon escient. Le principal apport d'Eliza est sans doute d'avoir montré qu'une machine est capable de simuler un comportement raisonnablement intelligent lorsque le champ de la conversation est assez étroit pour que le programmeur puisse cerner au préalable les mots et les concepts qui vont intervenir et les introduire dans un programme.

8
Graphisme

Un petit dessin vaut mieux qu'un long discours ...

Nous abordons maintenant le graphisme en Caml et illustrons l'emploi des types enregistrement à champs mutables. Accessoirement, nous réaliserons aussi de jolis dessins qu'on obtient grâce au graphisme « tortue », dans l'esprit du langage de programmation Logo.

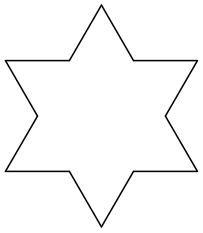
8.1 Fractales

Le but de ce chapitre, notre chef-d'œuvre graphique, est de tracer une courbe fractale très connue : le flocon de neige de von Koch. Une courbe fractale est une courbe très « tordue », c'est-à-dire comportant beaucoup de plis. Généralement définies récursivement à partir d'un motif de base, les fractales reproduisent à l'infini le motif à échelle de plus en plus réduite. Ainsi, quel que soit le niveau de détail où on l'examine, la fractale présente toujours son motif. Le flocon de von Koch est défini à partir du motif suivant :

Ce motif est ensuite reproduit sur chacun des côtés du motif de base à échelle réduite, ce qui donne :

La génération suivante sera donc :

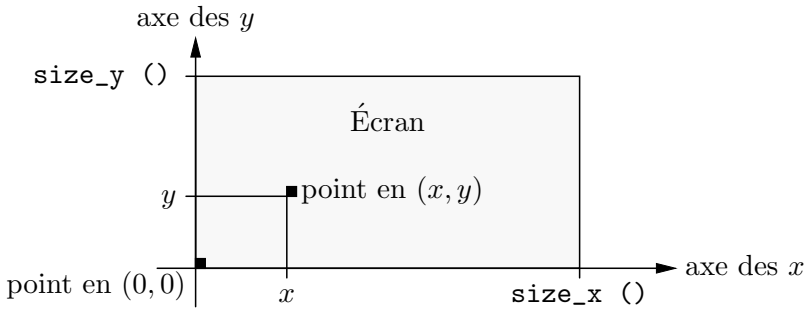
Ce mécanisme est reproductible à l'infini en mathématiques. Nous nous contenterons évidemment d'atteindre la résolution maximale de l'écran. Le flocon de von Koch proprement dit s'obtient simplement en reproduisant trois fois le motif sur les trois côtés d'un triangle équilatéral. La première génération donne donc le dessin ci-contre.



Afin de dessiner le flocon, nous allons implémenter un crayon programmable, version simplifiée de la tortue du langage Logo. Nous pourrons ensuite programmer ses déplacements sur l'écran pour qu'il trace le flocon.

8.2 Le graphisme de Caml

Le crayon se déplace dans le repère du graphisme de Caml. Ce repère a la forme suivante :



L'origine est donc en bas à gauche de l'écran. La taille de l'écran en x et en y est donnée par les primitives `size_x` et `size_y`.

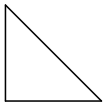
Le graphisme de Caml utilise les notions de point courant et de couleur de tracé courante. L'ordre `lineto x y` trace une ligne de la couleur courante, joignant le point courant au point de coordonnées `(x,y)`. Le point courant se trouve alors à l'extrémité du segment qu'on vient de tracer. L'ordre `moveto x y` permet de placer le point courant au point `(x,y)`. Un petit détail : les primitives graphiques ne sont pas accessibles directement ; il faut préalablement prévenir le système qu'on désire les utiliser par la directive `#open "graphics";;` (le dièse `#` fait partie de la directive, ce n'est pas le signe d'invite du système interactif). Nous considérerons pour l'instant cette directive comme une formule magique indispensable à l'utilisation du graphisme. Nous l'expliquerons plus tard, dans le chapitre 10.

```
# #open "graphics";;
```

De même, pour réaliser effectivement un dessin, il faut d'abord « ouvrir » l'écran graphique par un appel à la fonction prédéfinie `open_graph`.

```
# open_graph "";;
- : unit = ()
```

Dessinons maintenant un triangle rectangle isocèle dont l'angle droit a pour sommet l'origine du repère :



```
# moveto 0 0; lineto 0 50; lineto 50 0; lineto 0 0;;
- : unit = ()
```

On voit apparaître à l'écran le dessin en marge.

La bibliothèque graphique de Caml est indépendante de la machine sur laquelle tourne le système en ce qui concerne l'ensemble des primitives graphiques. En revanche, le nombre de points de l'écran et l'ensemble des couleurs possibles sont évidemment liés au matériel.

Pourquoi ne pas utiliser les primitives `lineto` et `moveto` pour dessiner le flocon ? Tout simplement parce qu'il faut leur préciser les coordonnées *absolues* des points du tracé, ce qui signifierait pour le flocon calculer les coordonnées de toutes les extrémités des segments qui forment le flocon. Cela paraît vraiment difficile. En revanche, nous décrirons facilement les mouvements *relatifs* du crayon, le faisant tourner et avancer tour à tour.

D'autre part le repère du graphisme de Caml ne comporte que des points dont les coordonnées sont entières, puisqu'il s'agit de points de l'écran. Cependant, pour des raisons de précision du tracé du crayon, nous maintiendrons les coordonnées du crayon en nombres décimaux. Il nous faudra donc arrondir les coordonnées du crayon avant d'effectuer ses déplacements sur l'écran.

8.3 Les nombres en représentation flottante

L'implémentation du crayon nécessite donc des « nombres à virgule » et plus précisément « à virgule flottante », qui sont fournis en Caml par le type prédéfini `float`. On les appelle « nombres flottants » en jargon informatique. Ils sont caractérisés par leur virgule mais la virgule dans les nombres se note avec un point en anglais ; naturellement c'est la notation avec un point qui s'est imposée en informatique. Le nombre 3,5 se note donc `3.5` en Caml. De même, les opérations de base sur les flottants, opérations arithmétiques et comparaisons, ont pour nom celui de l'opération correspondante sur les entiers suffixé par un point.

```
# 1.0 +. 2.0 *. 3.14159 >=. 18.9 /. 2.718;;
- : bool = true
```

En plus des quatre opérations, on dispose des fonctions transcendantes habituelles : l'exponentielle (`exp`), la fonction puissance (`power`) et les fonctions trigonométriques cosinus (`cos`), sinus (`sin`), tangente (`tan`), et leurs réciproques, logarithme (`log`), racine carrée (`sqrt`), arccosinus (`acos`), arcsinus (`asin`) et arctangente (`atan`). On dispose aussi de conversions entre entiers et flottants : `int_of_float` et `float_of_int`.

```
# sqrt 2.0;;
- : float = 1.41421356237
```

Nous définissons la fonction `round` pour arrondir les coordonnées du crayon à l'entier le plus proche :

```
# let round x =
    if x >=. 0.0 then int_of_float (x +. 0.5)
                 else int_of_float (x -. 0.5);;
round : float -> int = <fun>
```

8.4 Le crayon électronique

Le comportement du « crayon électronique » est le suivant :

- Le crayon trace dans une direction qu'on peut faire varier à la demande et que nous nommerons la « visée » du crayon.
- Il se déplace d'une distance qu'on lui indique, dans la direction de sa visée actuelle.

- En avançant le crayon ne laisse pas de trace s'il est levé et trace un trait s'il est baissé.

Pour gérer le crayon, il nous faut donc tenir à jour et faire évoluer son *état courant* : ses coordonnées, son angle de visée et le mode de tracé (crayon levé ou non). Cet état est décrit par un type définissant toutes les caractéristiques du crayon ; c'est donc un type « et » : un type enregistrement. Les coordonnées du crayon et son angle de visée sont des nombres flottants et le statut (levé ou non) du crayon est évidemment un booléen. Cela nous conduirait donc à définir le type `état` comme

```
type état =
   { x : float; y : float; visée : float; levé : bool };;
```

et l'état courant du crayon comme

```
let crayon =
  { x = 0.0; y = 0.0; visée = 0.0; levé = false };;
```

Cependant, ce type ne nous permet pas de faire évoluer le crayon. Or, nous n'avons qu'un seul crayon dont nous voulons faire varier dynamiquement les caractéristiques. Pour cela, il faut explicitement déclarer au système Caml que nous désirons modifier physiquement les champs de l'enregistrement qui modélise le crayon. Comme expliqué à la section 6.6, il suffit d'indiquer que les champs du type `état` sont modifiables, en faisant précéder les étiquettes correspondantes du mot-clé `mutable` lors de la définition du type.

```
# type état =
     { mutable x : float; mutable y : float;
       mutable visée : float; mutable levé : bool };;
Le type état est défini.
```

Le contrôleur de type nous autorisera maintenant à changer les valeurs des caractéristiques d'un objet du type `état`. La construction d'une valeur d'un enregistrement à champs mutables ne diffère pas du cas habituel. Nous définissons donc le crayon comme une donnée du type `état` par :

```
# let crayon = { x = 0.0; y = 0.0; visée = 0.0; levé = false };;
crayon : état = {x = 0.0; y = 0.0; visée = 0.0; levé = false}
```

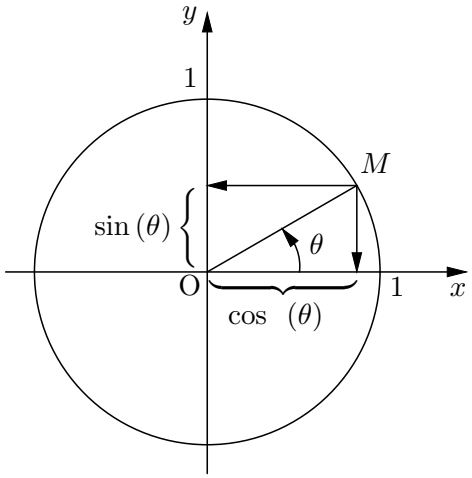
Tourner

Faire tourner le crayon consiste à changer son angle de visée, pour lui imprimer le nouveau cap. On utilise pour cela la modification physique d'un champ d'enregistrement, notée par une flèche vers la gauche, `<-`. Ainsi, la fonction qui permet de lever ou de baisser le crayon est simplement :

```
# let fixe_crayon b = crayon.levé <- b;;
fixe_crayon : bool -> unit = <fun>
```

L'angle de visée `crayon.visée` est exprimé en radians et suit les conventions du cercle trigonométrique des mathématiques : le zéro est à l'est et le crayon tourne dans le sens inverse de celui des aiguilles d'une montre. On rappelle que le cercle trigonométrique est le cercle de rayon 1 d'un repère orthonormé. Si l'angle θ est repéré par les demi-droites Ox et OM, alors les coordonnées (x, y) de M sont respectivement le cosinus et le sinus de l'angle θ.

Cependant, pour plus de commodité, les ordres de changement de cap donnés au crayon seront exprimés en degrés. La conversion est simple, puisqu'on a $Angle(\text{en radians}) = Angle(\text{en degrés}) \times \pi/180$. Après avoir nommé la valeur $\pi/180$ pour faire commodément les conversions de degrés en radians, nous définissons la fonction `tourne` qui change le cap du crayon.




```
# let pi_sur_180 =
      let pi = 4.0 *. (atan 1.0) in
      pi /. 180.0;;
pi_sur_180 : float = 0.0174532925199

# let tourne angle =
      crayon.visée <- (crayon.visée +. angle *. pi_sur_180);;
tourne : float -> unit = <fun>
```

Avancer

La primitive qui fait avancer le crayon se contente de calculer les déplacements du crayon nécessaires selon l'axe des abscisses et l'axe des ordonnées (`dx` et `dy`), à l'aide des formules trigonométriques de base, puis de modifier les coordonnées du crayon, et enfin de déplacer le crayon, soit en traçant (si le crayon est baissé) à l'aide de la primitive graphique `lineto`, soit sans tracer de trait (si le crayon est levé) en utilisant alors la primitive `moveto`.

```
# let avance d =
      let dx = d *. cos (crayon.visée)
      and dy = d *. sin (crayon.visée) in
      crayon.x <- crayon.x +. dx;
      crayon.y <- crayon.y +. dy;
      if crayon.levé
      then moveto (round crayon.x) (round crayon.y)
      else lineto (round crayon.x) (round crayon.y);;
avance : float -> unit = <fun>
```

Utilitaires d'initialisation du crayon

Pour simplifier le travail de l'utilisateur du crayon, le repère du crayon est proche de celui des mathématiques : l'origine est au centre de l'écran graphique. Les coordonnées de l'origine sont contenues dans deux constantes `zero_x` et `zero_y` qui valent donc respectivement `size_x ()`/2 et `size_y ()`/2.

On initialise donc le crayon en fixant ses coordonnées au centre de l'écran (`zéro_x`, `zéro_y`), en le faisant pointer vers l'est, en le baissant pour qu'il laisse une trace et en amenant le point courant du graphisme de Caml à la position actuelle du crayon. Enfin, et c'est le plus difficile, on efface l'écran. La fonction obtient cet effet en peignant tout l'écran avec la couleur du fond. L'écran forme un rectangle de coin inférieur gauche $(0, 0)$

et de coin supérieur droit (`size_x ()`, `size_y ()`). On utilise la fonction prédéfinie `fill_rect`, qui remplit un rectangle avec la couleur de tracé courante. Cette couleur est fixée par la fonction graphique `set_color`. Nous avons choisi les couleurs de fond et de tracé comme sur une feuille de papier, c'est-à-dire blanc pour le fond (couleur prédéfinie `white`) et noir pour les points tracés (couleur prédéfinie `black`).

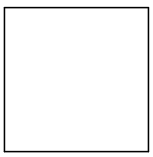
```
# let couleur_du_tracé = black;;
couleur_du_tracé : color = 0
# let couleur_du_fond = white;;
couleur_du_fond : color = 1
# let zéro_x = float_of_int ((size_x ()) / 2);;
zéro_x : float = 3000.0
# let zéro_y = float_of_int ((size_y ()) / 2);;
zéro_y : float = 2000.0


# let vide_écran () =
      set_color couleur_du_fond;
      fill_rect 0 0 (size_x ()) (size_y ());
      set_color couleur_du_tracé;
      crayon.x <- zéro_x;
      crayon.y <- zéro_y;
      crayon.visée <- 0.0;
      crayon.levé <- false;
      moveto (round crayon.x) (round crayon.y);;
vide_écran : unit -> unit = <fun>
```

8.5 Premiers dessins

Dessiner un carré avec le crayon est particulièrement simple : il suffit d'avancer quatre fois de la longueur du côté, en tournant à chaque fois d'un angle droit :

```
# let carré c =
      for i = 1 to 4 do
       avance c; tourne 90.0
      done;;
carré : float -> unit = <fun>
```

Nous initialisons le crayon, puis lançons le dessin.

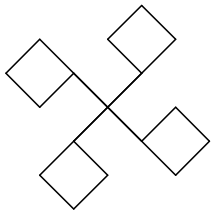


```
# vide_écran (); carré 75.0;;
- : unit = ()
```

Dessinons maintenant les ailes d'un moulin :

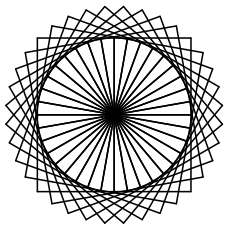
```
# let aile c =
      avance c; carré c; avance (-.c);;
aile : float -> unit = <fun>
```

```
# let ailes c =
      tourne 45.0;
      for i = 1 to 4 do aile c; tourne 90.0 done;;
ailes : float -> unit = <fun>
# vide_écran (); ailes 25.0;;
- : unit = ()
```



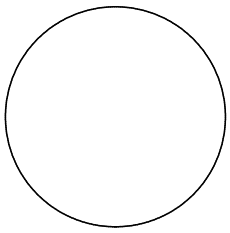
Une simili-rosace s'obtient en faisant tourner un carré sur son coin inférieur gauche :

```
# vide_écran ();
  for i = 1 to 36 do carré 40.0; tourne 10.0 done;;
- : unit = ()
```



Un cercle se dessine simplement à petits pas, en avançant un peu (d'un point, par exemple) et tournant un peu (d'un degré), pendant 360 degrés.

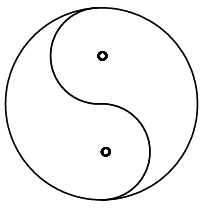
```
# let rond () =
    for i = 0 to 360 do
      avance 1.0; tourne 1.0
    done;;
rond : unit -> unit = <fun>
# vide_écran (); rond ();;
- : unit = ()
```



Il n'est pas difficile de définir une procédure générale pour dessiner un cercle de rayon R, ou des portions de cercle d'un rayon et d'un angle donnés. On doit maintenant calculer le pas du crayon : mais puisque le crayon parcourt la circonférence complète du cercle en 360 pas, on a $2 \times \pi \times R = 360 \times pas$ et le pas est donc $pas = \pi/180 \times R$:

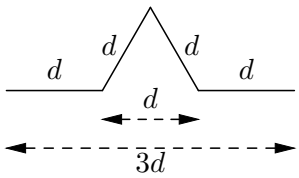
```
# let cercle r =
      let pas = r *. pi_sur_180 in
      for i = 0 to 360 do avance pas; tourne 1.0 done;;
cercle : float -> unit = <fun>
# let arc_gauche r angle =
      let pas = r *. pi_sur_180 in
      for i = 0 to angle do avance pas; tourne 1.0 done;;
arc_gauche : float -> int -> unit = <fun>
# let arc_droit r angle =
      let pas = r *. pi_sur_180 in
      for i = 0 to angle do avance pas; tourne (-. 1.0) done;;
arc_droit : float -> int -> unit = <fun>
```

```
# vide_écran (); cercle 50.0;
  arc_gauche 25.0 180; arc_droit 25.0 180;
  fixe_crayon true; tourne (-. 90.0); avance 25.0;
  fixe_crayon false; cercle 2.0;
  fixe_crayon true; avance 50.0;
  fixe_crayon false; cercle 2.0;;
- : unit = ()
```



8.6 Le flocon de von Koch

Pour le flocon de von Koch, il faut définir le motif de base, dont la taille dépend du paramètre c et de la génération n. Appelons `motif` cette procédure. À la génération 0, on avance simplement de la longueur du côté, c. Sinon il faut tracer les quatre morceaux du motif de base en tournant des bons angles. Ces quatre morceaux sont eux-mêmes des motifs, mais de la génération précédente. Chaque morceau est donc obtenu par un appel récursif à `motif` de la forme `motif (n-1) d`, où `d` est la longueur d'un morceau. Il ne nous reste plus qu'à déterminer cette longueur. Or il est facile de voir qu'en parcourant quatre fois la distance d en suivant le motif, on avance en fait linéairement de $3d$:



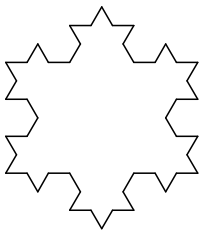
C'est donc que, si chacun des morceaux mesure d, la longueur totale du motif est $3d$. Comme cette longueur totale est `c`, on en déduit que $3d = c$ et donc $d = c/3$. On obtient la procédure suivante :

```
# let rec motif n c =
      if n = 0 then avance c else
       begin
        motif (n-1) (c /. 3.0); tourne 60.0;
        motif (n-1) (c /. 3.0); tourne (-120.0);
        motif (n-1) (c /. 3.0); tourne 60.0;
        motif (n-1) (c /. 3.0)
       end;;
motif : int -> float -> unit = <fun>
```

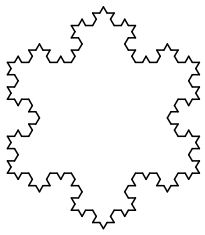
Le flocon à la $n^{\text{ième}}$ génération s'obtient simplement en traçant 3 fois le motif de génération n sur les trois côtés d'un triangle équilatéral.

```
# let flocon n c =
      for i = 1 to 3 do motif n c; tourne (-120.0) done;;
flocon : int -> float -> unit = <fun>
```

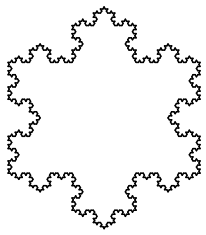
```
# vide_écran ();
  flocon 2 100.0;;
- : unit = ()
```



```
# vide_écran ();
  flocon 3 100.0;;
- : unit = ()
```



```
# vide_écran ();
  flocon 4 100.0;;
- : unit = ()
```



Impressionnant, n'est-ce pas ?

9
Syntaxe abstraite, syntaxe concrète

Un peu de recul permet souvent d'avancer ...

TOUS NOS OUTILS GRAPHIQUES sont maintenant en place et nous sommes prêts à transformer les programmes de pilotage du crayon électronique en un véritable langage de programmation. Ce chapitre introduit donc la notion de syntaxe abstraite et présente un exemple d'évaluation d'arbres de syntaxe abstraite. Nous aborderons aussi l'important concept d'analyse syntaxique. Nous esquisserons ainsi les étapes importantes d'une méthodologie de manipulation des données formelles qui nous servira dans toute la suite.

9.1 Présentation

Le but des programmes qui suivent est de simplifier encore l'utilisation du crayon pour qu'il puisse devenir un jeu d'enfant. Pour rendre le crayon utilisable par tous, nous aimerions assouplir le langage du crayon, de façon à :

1. avoir des ordres de déplacement plus explicites : avancer et reculer pour les déplacements, tourner à droite et à gauche pour les changements de l'angle de visée, lever et baisser la mine pour le tracé ;
2. définir une notion de répétition, pour éviter à l'utilisateur du crayon la manipulation des boucles « `for` » de Caml ;
3. pouvoir utiliser indifféremment des entiers ou des flottants dans les commandes du crayon, pour éviter les problèmes de « `.` » obligatoires dans les nombres (source d'erreurs innombrables pour les débutants).

On écrirait par exemple `répète 4 [avance 100 droite 90]` pour tracer un carré de côté 100, au lieu de l'expression `for i = 1 to 4 do avance 100.0; tourne 90.0 done`.

Si l'on tente de définir directement des procédures Caml pour obtenir ces nouvelles fonctionnalités, on se heurte très vite à des limites dues à la confusion entre le langage

d'implémentation (Caml) et le langage implémenté (celui du crayon). Expliquons-nous : l'ensemble des fonctions de manipulation du crayon définies dans le chapitre précédent, `avance`, `tourne`, `vide_écran` et `fixe_crayon`, forment un mini-langage graphique, complètement inclus dans Caml. En effet, il faut nécessairement utiliser les traits syntaxiques de Caml pour définir les dessins. Ainsi la définition du carré utilise explicitement les définitions globales (le `let`), la boucle `for`, la séquence (`;`) et les conventions lexicales de Caml pour les nombres flottants (`.0`) :

```
let carré c =
    for i = 1 to 4 do
     avance c; tourne 90.0
    done;;
```

Il n'y a rien là que de très normal après tout : nous n'avons fait qu'implémenter un ensemble de fonctions Caml. La question est de savoir comment aller plus loin et se débarrasser de la syntaxe de Caml pour notre langage graphique.

9.2 Le retard à l'évaluation

Voyons où et pourquoi nous nous heurtons à des difficultés réelles dans notre quête de la simplicité. Les fonctions `avance`, `recule`, `tourne_à_droite` et `tourne_à_gauche` sont faciles à définir.

```
# let recule d = avance (-. d)
  and tourne_à_droite a = tourne (-. a)
  and tourne_à_gauche = tourne;;
recule : float -> unit = <fun>
tourne_à_droite : float -> unit = <fun>
tourne_à_gauche : float -> unit = <fun>
```

Pour lever et baisser le crayon, on pourrait écrire :

```
# let baisse_le_crayon () = fixe_crayon false
  and lève_le_crayon () = fixe_crayon true;;
baisse_le_crayon : unit -> unit = <fun>
lève_le_crayon : unit -> unit = <fun>
```

Ce n'est pas parfait, car on oblige encore l'utilisateur à taper `baisse_le_crayon ()` pour baisser le crayon. Cette paire de parenthèses obligatoire n'est pas évidente pour tout le monde.

La grande difficulté est la fonction `répète`. On aimerait fournir à `répète` un nombre de répétitions à effectuer et une liste d'ordres graphiques à répéter, par exemple `répète 4 [avance 50.0; tourne 90.0]`. Naïvement, on définirait :

```
# let répète n l =
      for i = 1 to n do l done;;
répète : int -> 'a -> unit = <fun>
```

Malheureusement la liste `l` des ordres est complètement évaluée (et donc les ordres qu'elle contient sont exécutés) au moment de l'appel de la fonction `répète` ! Dans le corps de `répète` on ne fait que relire une liste de valeurs « rien », ce qui n'a pas grand effet.

```
# répète 4 [print_int 1; print_char '*'];;
*1*- : unit = ()
```

Le 1 s'affiche avant l'exécution de la boucle qui, elle, n'affiche rien du tout, puisqu'elle ne fait qu'évaluer quatre fois de suite la liste de « riens » `[(); ()]`, résultat de l'évaluation du deuxième argument de **`répète`**. D'ailleurs, le type de **`répète`** aurait pu nous mettre la puce à l'oreille : `int -> 'a -> unit` met bien en évidence que **`répète`** n'utilise pas vraiment son argument `l`, la liste des ordres, puisque cet argument n'est finalement soumis à aucune contrainte, pas même celle d'être une liste. Vous aurez aussi remarqué que le caractère `*` s'est imprimé avant l'entier `1` : les éléments d'une liste sont évalués dans un ordre non spécifié par le langage. Dans le cas présent, le compilateur a choisi l'ordre droite-gauche, ce qui ne convient pas du tout. Il va sans dire que vos programmes ne doivent pas reposer sur l'ordre d'évaluation que choisit aujourd'hui le compilateur de votre machine : la prochaine version du compilateur pourrait bien en choisir un autre. Il ne faut donc pas faire d'effets pendant la construction d'une donnée.

Cette solution naïve ne marche donc pas du tout. Pour obtenir l'effet désiré, il nous faudrait *retarder l'évaluation* de la liste d'ordres, par exemple en passant une fonction en argument, au lieu d'une liste :

```
# let répète n liste_d'ordres =
      for i = 1 to n do liste_d'ordres () done;;
répète : int -> (unit -> 'a) -> unit = <fun>
# répète 4 (function () -> print_int 1; print_char '*');;
1*1*1*1*- : unit = ()
```

Le résultat voulu est atteint, mais l'utilisation de **`répète`** devient extrêmement lourde — très « informatique », disons.

Conclusion : pour dépasser ce problème, il faut prendre du recul, c'est-à-dire manipuler les ordres graphiques non plus comme des fonctions de Caml, mais comme des *données*. Nous pourrons ainsi en maîtriser complètement l'évaluation. Nous définirons donc le type **`ordre`** des ordres graphiques et une fonction **`exécute_ordre`** qui les exécutera. La fonction **`répète`** prendra alors en argument une liste de valeurs du type **`ordre`**, qu'elle pourra exécuter autant de fois que nécessaire en utilisant la fonction **`exécute_ordre`**.

Nous résolvons de la même manière le problème des nombres en définissant un type **`nombre`** qui regroupe des valeurs flottantes et entières.

```
# type nombre =
    | Entier of int
    | Flottant of float;;
Le type nombre est défini.
```

Les opérations de base du crayon n'acceptant que des arguments flottants, il nous faut une fonction traduisant les nombres (valeurs du type **`nombre`**) en valeurs du type **`float`**.

```
# let flottant = function
   | Entier i -> float_of_int i
   | Flottant f -> f;;
flottant : nombre -> float = <fun>
```

9.3 L'évaluation des ordres du langage graphique

Nous définissons maintenant le type des ordres graphiques :

Constructeur	Ordre graphique représenté	Constructeur	Ordre graphique représenté
`Av`	avance	`Re`	recule
`Td`	tourne à droite	`Tg`	tourne à gauche
`Lc`	lève le crayon	`Bc`	baisse le crayon
`Ve`	vide l'écran	`Rep`	répétition d'une liste d'ordres

```
# type ordre =
     | Av of nombre | Re of nombre
     | Td of nombre | Tg of nombre
     | Lc | Bc
     | Ve
     | Rep of int * ordre list;;
Le type ordre est défini.
```

La fonction d'évaluation exécute les ordres graphiques en utilisant les fonctions graphiques du crayon. La seule subtilité, pour l'ordre `Rep`, consiste à itérer l'évaluation sur la liste des ordres à l'aide de la fonctionnelle `do_list`.

```
# let rec exécute_ordre = function
    | Av n -> avance (flottant n)
    | Re n -> avance (-. (flottant n))
    | Tg a -> tourne (flottant a)
    | Td a -> tourne (-. (flottant a))
    | Lc -> fixe_crayon true
    | Bc -> fixe_crayon false
    | Ve -> vide_écran ()
    | Rep (n, l) -> for i = 1 to n do do_list exécute_ordre l done;;
exécute_ordre : ordre -> unit = <fun>
```

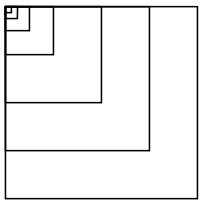
Si nous considérons qu'un programme du langage graphique n'est rien d'autre qu'une liste d'ordres, alors la fonction qui exécute les programmes n'est qu'une banale itération de la fonction `exécute_ordre` sur tous les ordres qui composent le programme.

```
# let exécute_programme l = do_list exécute_ordre l;;
exécute_programme : ordre list -> unit = <fun>
```

Nous pouvons maintenant écrire :

```
# let carré c = Rep (4, [Av c; Td (Entier 90)]);;
carré : nombre -> ordre = <fun>
```

```
# exécute_programme
   [Ve; carré (Entier 100); carré (Entier 75);
    carré (Entier 50); carré (Entier 25);
    carré (Flottant 12.5); carré (Flottant 6.25);
    carré (Flottant 3.125)];;
- : unit = ()
```



9.4 Syntaxe et sémantique

Nous avons pu implémenter `répète` et tous les nouveaux ordres graphiques, ainsi que des nombres comportant à la fois des entiers et des flottants, mais il semble que nous ayons vraiment perdu en lisibilité par rapport à la version originale ! Alors, échec ? Non, car il faut encore apprendre à dissocier l'entrée des données de leur évaluation. Nous avons réussi à écrire un programme qui réalise notre plan initial en ce qui concerne la sémantique : nous avons la fonction `répète` désirée et le comportement correct pour les nombres. Mais nous avons échoué en ce qui concerne la syntaxe, c'est-à-dire l'ergonomie de l'écriture des ordres graphiques. En effet, en définissant un type de données Caml des ordres graphiques, nous ne nous sommes pas dégagés de la syntaxe des programmes Caml. Ce que nous désirons maintenant, c'est écrire comme bon nous semble les ordres graphiques et que cette écriture, agréable pour nous, engendre une valeur Caml de type `ordre`. Par exemple, nous souhaitons écrire `répète 3 [avance 30 droite 90]`, mais évaluer la valeur Caml `Rep (3, [Av (Entier 30); Td (Entier 90)])`.

Nous allons donc écrire un programme qui réalise automatiquement cette transformation. À partir d'une chaîne de caractères en entrée, notre programme l'analysera pour produire en sortie un ordre graphique. Les programmes spécialisés dans ce genre de transformations s'appellent des *analyseurs syntaxiques*. La manière d'écrire les chaînes d'entrée légales est la *syntaxe concrète*, tandis que la valeur produite par l'analyseur syntaxique est la *syntaxe abstraite*. Par exemple, la syntaxe concrète `avance 30` a pour syntaxe abstraite `Av(Entier 30)`. La syntaxe abstraite de notre langage graphique est donc décrite par le type Caml `ordre`. Intuitivement, la syntaxe abstraite donne la signification de la phrase analysée d'une manière synthétique et indépendante des variations possibles de la syntaxe concrète. Par exemple, nous tolérerons les abréviations d'ordres graphiques, comme `av` au lieu de `avance`. L'expressions `av 30` aura donc la même syntaxe abstraite que `avance 30`, soit `Av (Entier 30)`. Ainsi, nos fonctions d'exécution des ordres graphiques n'auront même pas à savoir qu'il existe deux moyens de dire « avance », puisque leur argument sera exactement le même dans les deux cas. (C'est un exemple très simple de *normalisation* des données avant traitement.)

Nous résumons syntaxe concrète, syntaxe abstraite et sémantique des ordres graphiques dans le tableau suivant.

Syntaxe concrète Langage graphique Monde du crayon	Syntaxe abstraite Langage Caml Monde Caml	Sémantique (signification) Dessin à l'écran ou modification de l'état du crayon
`avance 30.0` `av 30.0`	`Av (Flottant 30.0)` `Av (Flottant 30.0)`	`avance 30.0` `avance 30.0`
`recule 10.0` `re 10`	`Re (Flottant 10.0)` `Re (Entier 10)`	`avance (-. 10.0)` `avance (-. 10.0)`
`gauche 10` `tg 10`	`Tg (Entier 10)` `Tg (Entier 10)`	`tourne 10.0` `tourne 10.0`
`droite 10` `td 10`	`Td (Entier 10)` `Td (Entier 10)`	`tourne (-. 10.0)` `tourne (-. 10.0)`

`lève_crayon`
`lc`
`baisse_crayon`
`bc`
	`Lc`
	`Lc`
	`Bc`
	`Bc`
		`fixe_crayon true`
		`fixe_crayon true`
		`fixe_crayon false`
		`fixe_crayon false`

`vide_écran` `ve`	`Ve` `Ve`	`vide_écran ()` `vide_écran ()`

Pour la répétition des ordres, nous avons besoin des suites d'ordres séparés par des blancs et placés entre crochets `[` et `]`, comme pour les listes de Caml.

Syntaxe concrète	Syntaxe abstraite	Sémantique (signification)
Ordres entre crochets	Liste d'ordres	Exécution des ordres de la liste
`[av 30 td 90]`	`[Av(Entier 30);` `Td(Entier 90)]`	Itération avec `do_list` de la fonction `exécute_ordre`
Répétition d'ordres	Ordre `Rep`	Une boucle « for » de Caml

`répète 3`
`[av 30 td 90]`
	`Rep(3,`
	`[Av(Entier 30);`
	`Td(Entier 90)])`
		Une boucle « for » qui exécute trois fois la liste des ordres

`rep 3 [av 30 td 90]`	`Rep(3,`
	`[Av(Entier 30);`
	`Td(Entier 90)])`
		Une boucle « for » qui exécute trois fois la liste des ordres

9.5 Notions d'analyses syntaxique et lexicale

L'analyse syntaxique est un problème difficile. Elle correspond à l'analyse grammaticale de la langue française, que vous devez déjà avoir appris à redouter. Cependant, lorsqu'un être humain analyse une phrase française, une étape extrêmement simple et intuitive est passée sous silence, tellement elle va de soi : la séparation de la phrase en mots ! C'est cependant une phase non triviale en informatique : l'analyse *lexicale*. Lorsque l'analyseur lexical a reconnu les mots de la phrase (qu'on appelle aussi lexèmes ou unités lexicales), l'analyseur *syntaxique* regroupe ces mots en phrases, selon les règles de la grammaire (la syntaxe concrète) du langage à reconnaître.

Prenons notre exemple favori : `avance 30`. L'analyseur lexical commence par reconnaître que la phrase est formée de deux mots, « `avance` » et « `30` ». Le lexème « `30` » est un entier car c'est une suite ininterrompue de chiffres. Notez que l'analyseur lexical

ne se laissera pas démonter par une entrée un peu différente comme `avance  30` (deux espaces entre `avance` et `30`), ou même

```
avance
  30
```

qui sépare les deux mots par un saut de ligne et deux espaces. Le travail de l'analyseur syntaxique sera énormément simplifié par cette normalisation de l'entrée faite par la phase d'analyse lexicale. Dans notre cas, l'analyseur syntaxique examinera les règles de la grammaire pour essayer de former une phrase avec les deux mots fournis par l'analyseur lexical. L'une de ces règles est justement

```
[< 'Mot "avance"; nombre n >] -> Av n
```

qui signifie : si l'on rencontre le mot (`'Mot`) `avance`, suivi (`;`) d'un nombre `n` (`nombre n`), alors c'est une phrase légale (`->`), dont la syntaxe abstraite est `Av n`. L'analyseur syntaxique nous renverra donc automatiquement la valeur Caml `Av n`, que nous ne voulions justement pas avoir besoin d'écrire, à partir du format d'entrée qui nous convient.

Dans la section suivante, nous allons voir en détails les analyseurs syntaxique et lexical du langage graphique.

9.6 Analyse lexicale et syntaxique

Les flux

Pour programmer l'analyse lexicale et l'analyse syntaxique du langage, nous allons utiliser une structure de données Caml que nous n'avons pas encore vue : les *flux* (*streams* en anglais). Tout comme les listes, les flux sont des suites de valeurs du même type. Le type d'un flux est t `stream`, où t est le type des éléments du flux. Ainsi, un flux d'entiers a le type `int stream`, de la même manière qu'une liste d'entiers a le type `int list`. Nous programmerons l'analyseur lexical comme une fonction qui prend un flux de caractères en entrée (type `char stream`) et produit un flux de lexèmes en sortie (type `lexème stream`). Nos lexèmes comprennent des entiers, des flottants, des mots simples (suites de caractères commençant par une lettre) et des symboles, c'est-à-dire des caractères qui ne sont ni chiffres, ni lettres (par exemple le point « . »). Les lexèmes sont donc décrits par le type concret suivant :

```
# type lexème =
     | Mot of string
     | Symbole of char
     | Constante_entière of int
     | Constante_flottante of float;;
Le type lexème est défini.
```

De même, l'analyseur syntaxique se présentera sous la forme d'une fonction qui prend un flux de lexèmes en entrée (type `lexème stream`) et produit un arbre de syntaxe (type `ordre`) en sortie. En composant les deux, nous obtiendrons une fonction qui transforme un flux de caractères (syntaxe concrète) en un arbre (syntaxe abstraite).

Nous produirons le flux de caractères initial à partir d'une chaîne de caractères, grâce à la fonction prédéfinie `stream_of_string`, qui transforme une chaîne en le flux des caractères qui la composent :

```
# let flux_car = stream_of_string "Vive Caml!";;
flux_car : char stream = <abstr>
```

Une autre manière de construire des flux est d'énumérer leurs éléments à l'intérieur de « crochets pointus » `[< ... >]`.

```
# let flux_ent = [< '2; '3; '5; '7 >];;
flux_ent : int stream = <abstr>
```

Nous verrons plus tard d'autres manières de construire des flux. Pour l'instant, notons simplement que chaque élément du flux est introduit par le caractère `'` (apostrophe) et que le système interactif ne sait pas imprimer le contenu des flux.

Pourquoi faire appel à des flux, au lieu d'utiliser des listes ? Les flux diffèrent des listes sur deux points importants, qui rendent les flux mieux adaptés aux problèmes d'analyse lexico-syntaxique. Première différence : l'accès dans un flux est destructif. Cela signifie que lorsqu'on consulte le premier élément d'un flux, cet élément est aussitôt retiré du flux et remplacé par l'élément suivant. On le voit bien à l'aide de la fonction prédéfinie `stream_next`, qui renvoie le premier élément d'un flux :

```
# stream_next flux_car;;
- : char = 'V'
# stream_next flux_car;;
- : char = 'i'
```

Ce comportement de lecture destructrice est bien adapté à l'analyse lexico-syntaxique : en général, les analyseurs lisent une seule fois leur entrée, sans jamais revenir en arrière ; il ne sert donc à rien de conserver les éléments de l'entrée une fois qu'ils ont été lus.

La deuxième particularité des flux est que les éléments contenus dans un flux ne sont pas évalués en bloc quand le flux est créé, mais petit à petit, au fur et à mesure qu'on y accède. En particulier, lorsqu'on construit le flux des caractères provenant d'un fichier, ce dernier n'est pas lu tout entier en mémoire : le flux ne contient en mémoire que le caractère courant et va chercher le prochain caractère sur le disque lorsqu'on en a besoin. Ce comportement est économique en mémoire, en particulier quand le fichier est gros. En termes savants, ce procédé s'appelle *évaluation paresseuse*, par analogie avec la stratégie bien connue consistant à remettre à demain ce qu'il n'est pas absolument nécessaire de faire aujourd'hui.

Analyse lexicale

L'analyse lexicale consiste à transformer un flux de caractères en le flux des lexèmes correspondants, avons-nous dit. Nous allons nous fixer un but plus modeste : construire une fonction `lire_lexème` qui prend un flux de caractères en argument, reconnaît le premier lexème au début de ce flux et renvoie l'objet de type `lexème` qui le décrit. Comme la lecture sur les flux est destructrice, `lire_lexème` aura retiré du flux les caractères du lexème ; il suffira donc de rappeler `lire_lexème` sur le même flux pour lire le lexème suivant.

La manière la plus pratique de lire un flux n'est pas d'appeler `stream_next`, mais de faire du filtrage sur les premiers éléments du flux. Voici par exemple une fonction qui supprime tous les caractères blancs (espaces, tabulations et retours chariot) en tête d'un flux.

```
# let rec saute_blancs flux =
    match flux with
    | [< ' ' ' ' >] -> saute_blancs flux  (* ' ' est l'espace *)
    | [< ' '\t' >] -> saute_blancs flux (* '\t' est la tabulation *)
    | [< ' '\n' >] -> saute_blancs flux (* '\n' est la fin de ligne *)
    | [< >] -> ();;
saute_blancs : char stream -> unit = <fun>
```

Comme les expressions de flux, les motifs sur les flux sont notés entre crochets pointus `[< ... >]`. Ils filtrent le début du flux et non pas le flux tout entier. Ainsi, le motif `[< >]` filtre n'importe quel flux et non pas uniquement les flux vides comme on pourrait le croire. De même, le motif `[< 'm >]`, où m est un motif quelconque, filtre tous les flux dont le premier élément est filtré par m, même si le flux contient plus d'un élément.

La définition de `saute_blancs` se lit donc : « si le premier caractère de `flux` est un espace (motif `' '`), alors se rappeler récursivement sur `flux` ; faire de même si le premier caractère de `flux` est le caractère de tabulation (motif `'\t'`) ou le caractère fin de ligne (motif `'\n'`) ; dans tous les autres cas, renvoyer `()`. » On pourrait craindre que les appels récursifs à `saute_blancs` ne terminent pas, puisqu'on se rappelle sur le même argument (`flux`) que celui qu'on a reçu. Ce n'est pas vrai, car le flux a été physiquement modifié entre-temps : dès qu'un des trois premiers motifs s'applique, le premier élément du flux est enlevé de ce dernier. L'appel récursif de `saute_blancs` s'effectue donc sur le reste du flux, comme désiré. En revanche, lorsque les trois premiers motifs ne s'appliquent pas (parce que le premier caractère du flux n'est pas un blanc), le premier caractère du flux n'est pas supprimé : le dernier motif étant vide, il ne consomme aucun élément du flux.

Incidemment, une manière plus compacte d'écrire la fonction `saute_blancs` est de regrouper les trois premiers cas, comme ceci :

```
# let rec saute_blancs flux =
    match flux with
    | [< ' (' ' | '\t' | '\n') >] -> saute_blancs flux
    | [< >] -> ();;
saute_blancs : char stream -> unit = <fun>
```

La barre verticale `|` dans les motifs signifie « ou ». Le motif `(' ' | '\t' | '\n')` se lit donc comme « un espace, une tabulation, ou un caractère de fin de ligne ». Poursuivons dans la même veine par la fonction qui lit des entiers.

```
# let rec lire_entier accumulateur flux =
    match flux with
    | [< '('0'..'9' as c) >] ->
        lire_entier (10 * accumulateur + int_of_char c - 48) flux
    | [< >] -> accumulateur;;
lire_entier : int -> char stream -> int = <fun>
```

Le motif `'0'..'9'` filtre tous les caractères entre `0` et `9` dans le jeu de caractères ASCII, c'est-à-dire tous les chiffres. C'est une abréviation pour `'0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'`. Que dit la fonction `lire_entier` ? « Si le premier caractère de `flux` est un chiffre, alors se rappeler récursivement pour lire la suite du nombre entier, sinon s'arrêter et renvoyer l'entier lu jusqu'ici. » Le paramètre `accumulateur` est la valeur entière du nombre que représentent les chiffres lus jusqu'ici. Les chiffres ont les codes 48

à 57 en ASCII ; donc, `int_of_char c - 48` est l'entier entre zéro et neuf qui représente le chiffre `c`. L'appel récursif sur `10 * accumulateur + int_of_char c - 48` revient bien à introduire le chiffre `c` à droite du nombre `accumulateur`. Exemple d'exécution :

```
# let flux_car = stream_of_string "123/456";;
flux_car : char stream = <abstr>
# lire_entier 0 flux_car;;
- : int = 123
# stream_next flux_car;;
- : char = '/'
# lire_entier 900 flux_car;;
- : int = 900456
```

Pour lire les nombres flottants, nous avons besoin d'une fonction similaire à `lire_entier`, mais lisant les chiffres à droite de la virgule et non plus à gauche.

```
# let rec lire_décimales accumulateur échelle flux =
    match flux with
    | [< '('0'..'9' as c) >] ->
        lire_décimales
          (accumulateur +.
             float_of_int(int_of_char c - 48) *. échelle)
          (échelle /. 10.0) flux
    | [< >] -> accumulateur;;
lire_décimales : float -> float -> char stream -> float = <fun>
```

Un exemple devrait mieux nous faire comprendre qu'un long discours :

```
# lire_décimales 123.4 0.01 (stream_of_string "56789");;
- : float = 123.456789
```

Dernière étape avant la lecture des lexèmes : la lecture d'un mot. Précisons qu'un mot, ici, est une suite de lettres majuscules ou minuscules (de `A` à `Z` et de `a` à `z`, plus quelques lettres accentuées).

```
# let tampon = make_string 16 '-';;
tampon : string = "----------------"
# let rec lire_mot position flux =
    match flux with
    | [< '('A'..'Z' | 'a'..'z' | 'é' | 'è' | '_' as c) >] ->
        if position < string_length tampon
         then tampon.[position] <- c;
        lire_mot (position + 1) flux
    | [< >] ->
        sub_string tampon 0 (min position (string_length tampon));;
lire_mot : int -> char stream -> string = <fun>
```

La chaîne `tampon` sert à accumuler les caractères du mot en cours de lecture. Le paramètre `position` de `lire_mot` est le numéro du caractère de `tampon` où il faut stocker le prochain caractère du mot. (On rappelle que la construction s`.[`n`] <-` c remplace le $n^{\text{ième}}$ caractère de la chaîne s par le caractère c.) Paraphrasons `lire_mot`. « Si le premier caractère de `flux` est une lettre, alors le stocker dans `tampon` à l'endroit désigné par `position`, pourvu qu'il reste de la place libre dans `tampon`, et se rappeler récursivement pour lire la suite. Sinon, extraire de `tampon` le mot lu jusqu'ici et le

renvoyer. » (L'appel `sub_string` s `0` n renvoie une chaîne constituée des n premiers caractères de s.)

La fonction `lire_lexème` tant attendue se contente de regarder le premier caractère non blanc du flux et selon que c'est une lettre, un chiffre ou autre chose, d'appeler une des fonctions de lecture précédemment définies.

```
# let lire_lexème flux =
    saute_blancs flux;
    match flux with
    | [< '(‘A‘..‘Z‘ | ‘a‘..‘z‘ | ‘é‘ | ‘è‘ as c) >] ->
        tampon.[0] <- c;
        Mot(lire_mot 1 flux)
    | [< '(‘0‘..‘9‘ as c) >] ->
        let n = lire_entier (int_of_char c - 48) flux in
        begin match flux with
        | [< ''.‘ >] ->
            Constante_flottante
              (lire_décimales (float_of_int n) 0.1 flux)
        | [< >] -> Constante_entière n
        end
    | [< 'c >] -> Symbole c;;
lire_lexème : char stream -> lexème = <fun>
```

En guise de test :

```
# let flux_car = stream_of_string "123bonjour   ! 45.67";;
flux_car : char stream = <abstr>
# lire_lexème flux_car;;
- : lexème = Constante_entière 123
# lire_lexème flux_car;;
- : lexème = Mot "bonjour"
# lire_lexème flux_car;;
- : lexème = Symbole ‘!‘
# lire_lexème flux_car;;
- : lexème = Constante_flottante 45.67
```

Pour finir, il reste à construire le flux des lexèmes lus.

```
# let rec analyseur_lexical flux =
   match flux with
   | [< lire_lexème l >] -> [< 'l; analyseur_lexical flux >]
   | [< >] -> [< >];;
analyseur_lexical : char stream -> lexème stream = <fun>
```

Cette fonction utilise deux nouvelles opérations sur les flux. Premièrement, l'ajout d'un élément x en tête d'un flux f se note `[< '`x`; `f` >]`, sans apostrophe devant le f. De même, la concaténation de deux flux f_1 et f_2 se note `[< `f_1`; `f_2`>]`, sans apostrophes du tout. Le point-virgule à l'intérieur des crochets pointus peut se lire comme l'opérateur de concaténation de flux ; l'apostrophe, comme l'opérateur qui prend une valeur x et renvoie le flux à un seul élément x.

Autre opération nouvelle sur les flux : on peut, lors d'un filtrage sur un flux, appeler une fonction d'analyse depuis l'intérieur du motif. Ceci se note `[< lire_lexème l >]`, sans apostrophe avant `lire_lexème`, et signifie : « appliquer la fonction `lire_lexème`

au flux en cours de filtrage (ici, `flux`) ; si cette fonction réussit, appeler `l` son résultat et continuer le filtrage ; si cette fonction échoue, essayer le prochain cas du filtrage ».

La fonction `lire_lexème` échoue quand aucun des cas de son filtrage ne s'applique, c'est-à-dire quand on lui passe un flux vide. Si `flux` est vide, l'appel à `lire_lexème` échoue donc et on passe au deuxième cas de `analyseur_lexical` qui renvoie le flux vide. Si `flux` n'est pas vide, l'appel à `lire_lexème` réussit et extrait de `flux` son premier lexème ; la représentation de ce lexème transite par la variable `l` et est ajoutée en tête du flux de lexèmes obtenus par appel récursif de `analyseur_lexical` sur le reste du flux.

Vous n'êtes toujours pas convaincu ? Essayons donc.

```
# let flux_lexèmes =
      analyseur_lexical (stream_of_string "123bonjour   ! 45.67");;
flux_lexèmes : lexème stream = <abstr>
# stream_next flux_lexèmes;;
- : lexème = Constante_entière 123
# stream_next flux_lexèmes;;
- : lexème = Mot "bonjour"
# stream_next flux_lexèmes;;
- : lexème = Symbole '!'
# stream_next flux_lexèmes;;
- : lexème = Constante_flottante 45.67
```

Analyse syntaxique

Puisque l'analyseur lexical renvoie un flux de lexèmes, l'analyseur syntaxique est une fonction qui prend en argument un flux de lexèmes et le transforme en une valeur plus structurée.

Nous commencerons par une partie de l'analyseur syntaxique qui se contente de lire un nombre, soit entier soit flottant, et d'en faire une valeur de type `nombre`. Cette fonction va donc envisager deux cas : si le flux de lexèmes commence par un lexème `Constante_entière i` alors elle fabrique le nombre `Entier i` ; si le flux de lexèmes commence par un lexème `Constante_flottante f` alors la fonction renvoie le nombre `Flottant f`. Tous les autres lexèmes produiront une erreur, se traduisant par le déclenchement de l'exception prédéfinie `Parse_failure`.

```
# let nombre = function
    | [< 'Constante_entière i >] -> Entier i
    | [< 'Constante_flottante f >] -> Flottant f;;
nombre : lexème stream -> nombre = <fun>
```

Par exemple :

```
# let flux_lexèmes =
      analyseur_lexical (stream_of_string "123 1.05 fini");;
flux_lexèmes : lexème stream = <abstr>
# nombre flux_lexèmes;;
- : nombre = Entier 123
# nombre flux_lexèmes;;
- : nombre = Flottant 1.05
```

```
# nombre flux_lexèmes;;
Exception non rattrapée: Parse_failure
```

La fonction qui analyse les ordres n'est guère plus compliquée. Pour les ordres sans argument, elle se contente de chercher le mot correspondant comme premier lexème du flux. Ainsi, pour analyser l'ordre `baisse_crayon`, on aura la clause `[< 'Mot "baisse_crayon" >] -> Bc`.

Pour les ordres avec argument numérique, on commence par détecter l'ordre, puis on appelle l'analyseur des nombres, la fonction `nombre` précédente. Cet appel a lieu encore une fois dans la partie filtre de la clause. Ainsi, la clause `[< 'Mot "avance"; nombre n >] -> Av n` se paraphrase en : si le premier lexème du flux est le mot `avance` et que la fonction `nombre` analyse avec succès le lexème suivant en renvoyant la valeur `n`, alors cette clause est sélectionnée et l'on renvoie l'ordre `Av n`.

Pour l'ordre `répète`, on cherche l'entier indiquant le nombre de répétitions, puis on cherche une liste d'ordres en appelant l'analyseur spécialisé `liste_d'ordres` qui reconnaît une succession d'ordres entourés de crochets, conformément à la syntaxe concrète que nous avons choisie.

L'analyseur `liste_d'ordres` attend donc un caractère crochet ouvrant, `Symbole '['`, puis appelle lui aussi un analyseur spécialisé dans la reconnaissance des successions d'ordres ; enfin, `liste_d'ordres` vérifie que la liste se termine bien par un crochet fermant, `Symbole ']'`.

```
# let rec ordre = function
    | [< 'Mot "baisse_crayon" >] -> Bc
    | [< 'Mot "bc" >] -> Bc
    | [< 'Mot "lève_crayon" >] -> Lc
    | [< 'Mot "lc" >] -> Lc
    | [< 'Mot "vide_écran" >] -> Ve
    | [< 'Mot "ve" >] -> Ve
    | [< 'Mot "avance"; nombre n >] -> Av n
    | [< 'Mot "av"; nombre n >] -> Av n
    | [< 'Mot "recule"; nombre n >] -> Re n
    | [< 'Mot "re"; nombre n >] -> Re n
    | [< 'Mot "droite"; nombre n >] -> Td n
    | [< 'Mot "td"; nombre n >] -> Td n
    | [< 'Mot "gauche"; nombre n >] -> Tg n
    | [< 'Mot "tg"; nombre n >] -> Tg n
    | [< 'Mot "répète"; 'Constante_entière n;
         liste_d'ordres l >] -> Rep (n,l)
    | [< 'Mot "rep"; 'Constante_entière n;
         liste_d'ordres l >] -> Rep (n,l)
  and liste_d'ordres = function
    | [< 'Symbole '['; suite_d'ordres l; 'Symbole ']' >] -> l
  and suite_d'ordres = function
    | [< ordre ord; suite_d'ordres l_ord >] -> ord::l_ord
    | [< >] -> [];;
ordre : lexème stream -> ordre = <fun>
liste_d'ordres : lexème stream -> ordre list = <fun>
suite_d'ordres : lexème stream -> ordre list = <fun>
```

La fonction `suite_d'ordres` est la plus complexe. Elle comprend deux filtres :

- `[< ordre ord; suite_d'ordres l_ord >]` : on s'attend à trouver un ordre `ord`, éventuellement suivi d'autres ordres, que la fonction `suite_d'ordres` placerait dans une liste `l_ord`.
- `[< >]` : dans tous les autres cas, il n'y a plus d'ordres à lire et la fonction renvoie la liste vide. C'est évidemment ce cas qui arrête la récursion de la fonction. Remarquez que cette syntaxe autorise la répétition d'une liste d'ordres vide.

Finalement, un programme est une suite d'ordres terminée par un point.

```
# let analyse_programme = function
    | [< suite_d'ordres l; 'Symbole `.` >] -> l;;
analyse_programme : lexème stream -> ordre list = <fun>
```

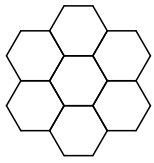
Nous définissons maintenant une fonction de lecture de programmes du langage graphique, qui lit un programme dans une chaîne de caractères et rend la liste des ordres correspondants.

```
# let lire_code chaîne =
      analyse_programme
        (analyseur_lexical (stream_of_string chaîne));;
lire_code : string -> ordre list = <fun>
# lire_code "répète 4 [avance 100 droite 90].";;
- : ordre list = [Rep (4, [Av (Entier 100); Td (Entier 90)])]
```

On combine facilement la lecture et l'exécution, obtenant enfin la fonction d'exécution d'une chaîne de caractères recherchée.

```
# let logo chaîne =
      exécute_programme (lire_code chaîne);;
logo : string -> unit = <fun>
```

En guise de test, imbriquons deux ordres `répète` pour faire se déplacer un hexagone.



```
# logo "ve répète 6
              [td 60 répète 6 [av 15 tg 60] av 15].";;
- : unit = ()
```

9.7 Ajout des procédures

Dans la section précédente, nous sommes parvenus à nous libérer de la syntaxe de Caml. Malheureusement, nous avons ainsi perdu la possibilité de définir des procédures, comme nous le faisions par exemple pour dessiner les ailes d'un moulin en utilisant la procédure `carré`. Nous allons donc étendre notre langage graphique avec une construction qui permette de définir des procédures.

La syntaxe abstraite des expressions du langage

Avec l'introduction des procédures et de leurs paramètres, nous sommes obligés d'introduire la notion d'expression dans le langage. En effet, l'argument d'un ordre, `avance` par exemple, ne se réduit plus à une constante : ce peut être maintenant l'argument d'une procédure. Par exemple, pour définir la procédure qui dessine un

carré de côté `c` nous écrirons une répétition où l'ordre `avance` a pour argument la variable `c` :

```
pour carré :c
     répète 4 [avance :c td 90].
```

La syntaxe concrète que nous employons s'inspire de celle du langage Logo : les noms de variables sont précédés d'un signe « deux points », comme dans « `:x` », et les définitions de procédures sont introduites par le mot `pour`.

Tous les ordres qui avaient précédemment un argument numérique auront maintenant une expression en argument. Nous étendons un peu ce langage des expressions pour autoriser les quatre opérations élémentaires. On trouvera donc dans le type des expressions, outre les nombres et les variables représentés par les constructeurs `Constante` et `Variable`, les quatre constructeurs `Somme`, `Produit`, `Différence` et `Quotient`. Ces constructeurs prennent en argument une paire d'expressions qui représente les deux opérandes.

```
# type expression =
     | Constante of nombre
     | Somme of expression * expression
     | Produit of expression * expression
     | Différence of expression * expression
     | Quotient of expression * expression
     | Variable of string;;
Le type expression est défini.
```

L'évaluateur des expressions

Nous devons évaluer nous-mêmes les opérations sur les nombres. C'est très simple : lorsque les deux nombres sont des entiers, il suffit d'appeler la primitive correspondante de Caml sur les entiers ; sinon, on transforme les deux nombres en flottants et l'on appelle la primitive correspondante, sur les flottants cette fois.

```
# let ajoute_nombres = function
    | (Entier i, Entier j) -> Entier (i + j)
    | (n1, n2) -> Flottant (flottant n1 +. flottant n2)
  and soustrais_nombres = function
    | (Entier i, Entier j) -> Entier (i - j)
    | (n1, n2) -> Flottant (flottant n1 -. flottant n2)
  and multiplie_nombres = function
    | (Entier i, Entier j) -> Entier (i * j)
    | (n1, n2) -> Flottant (flottant n1 *. flottant n2)
  and divise_nombres = function
    | (Entier i, Entier j) -> Entier (i / j)
    | (n1, n2) -> Flottant (flottant n1 /. flottant n2)
  and compare_nombres = function
    | (Entier i, Entier j) -> i >= j
    | (n1, n2) -> (flottant n1 >=. flottant n2);;
ajoute_nombres : nombre * nombre -> nombre = <fun>
soustrais_nombres : nombre * nombre -> nombre = <fun>
multiplie_nombres : nombre * nombre -> nombre = <fun>
divise_nombres : nombre * nombre -> nombre = <fun>
```

```
compare_nombres : nombre * nombre -> bool = <fun>
```

L'évaluateur des expressions lui-même est un peu plus complexe. En effet, il a un argument supplémentaire `env`, qui sert à retrouver la valeur courante d'une variable. C'est ce qu'on nomme un *environnement*. L'environnement sert à enregistrer les liaisons des variables à leur valeur, calculée lors de leur définition. Les liaisons sont modélisées par des paires (nom de variable, valeur associée), tandis que l'environnement se représente comme une liste de telles liaisons. L'environnement a donc la structure d'une liste d'association, structure de données que nous avons déjà utilisée pour modéliser la base de données de Camélia au chapitre 7. On ajoute donc une liaison à l'environnement en ajoutant une paire (nom, valeur) en tête de la liste qui représente l'environnement et l'on trouve la valeur associée à un nom de variable par une simple recherche à l'aide de la fonction `assoc`. (La fonction `assoc` est une fonction prédéfinie de recherche dans les listes d'associations, analogue à la fonction `associé_de` de la section 7.3.)

```
# let rec valeur_expr env = function
    | Constante n -> n
    | Somme (e1, e2) ->
        ajoute_nombres (valeur_expr env e1, valeur_expr env e2)
    | Produit (e1, e2) ->
        multiplie_nombres (valeur_expr env e1, valeur_expr env e2)
    | Différence (e1, e2) ->
        soustrais_nombres (valeur_expr env e1, valeur_expr env e2)
    | Quotient (e1, e2) ->
        divise_nombres (valeur_expr env e1, valeur_expr env e2)
    | Variable s -> assoc s env;;
valeur_expr : (string * nombre) list -> expression -> nombre = <fun>
```

Les ordres du langage

Les ordres s'enrichissent de trois nouvelles constructions : l'ordre `Stop` qui arrête une procédure, l'ordre `Exécute` qui exécute une procédure en l'appliquant à la liste de ses arguments et enfin une construction conditionnelle, `Si`. L'ordre `Si` exécute une liste d'ordres ou une autre, selon qu'une condition est remplie. Pour simplifier, la condition est réduite à la comparaison `>=` sur les nombres.

Le nouveau type `ordre` comprend donc les même constructeurs de base que l'ancien, `Av`, `Re`, ... , `Rep`, et les trois nouveaux constructeurs `Stop`, `Si` et `Exécute`.

```
# type ordre =
     | Av of expression | Re of expression
     | Td of expression | Tg of expression
     | Lc | Bc
     | Ve
     | Rep of expression * ordre list
     | Stop
     | Si of expression * expression * ordre list * ordre list
     | Exécute of string * expression list;;
Le type ordre est défini.
```

La définition des procédures et l'exécution des ordres

La construction `pour` permet de définir de nouvelles procédures. Lorsqu'une procédure est définie, son nom est enregistré avec son corps et la liste de ses arguments dans la liste d'association globale `procédures_définies`. De même que pour les variables, on recherche donc le corps et la liste d'arguments d'une procédure en cherchant sa valeur associée dans la liste des procédures déjà définies.

```
# type procédure = {paramètres : string list; corps : ordre list};;
Le type procédure est défini.
# let procédures_définies = ref ([] : (string * procédure) list);;
procédures_définies : (string * procédure) list ref = ref []
# let définit_procédure (nom, proc as liaison) =
      procédures_définies := liaison :: !procédures_définies
  and définition_de nom_de_procédure =
      assoc nom_de_procédure !procédures_définies;;
définit_procédure : string * procédure -> unit = <fun>
définition_de : string -> procédure = <fun>
```

L'exécution des ordres

Comme la fonction d'évaluation des expressions, la fonction d'exécution des ordres doit gérer un environnement, puisqu'elle est chargée de lier les arguments des procédures lorsqu'on les appelle. L'exécution des ordres simples ne change pas : on applique toujours les fonctions de base du crayon. La seule différence consiste à évaluer l'expression argument de l'ordre en appelant `valeur_expr` dans l'environnement courant. Par exemple, pour `Av e`, on appellera comme auparavant la fonction `avance` avec pour argument le flottant obtenu par l'évaluation de `e`, c'est-à-dire `valeur_expr env e`. L'ordre `répète` prend maintenant une expression en argument : cette expression est évaluée et retourne une valeur de type `nombre`. Cependant ce nombre n'est pas forcément un entier ; dans le cas où c'est une valeur flottante, deux options sont possibles : prendre la décision d'arrondir le `nombre` à l'entier le plus proche (répéter 3.1 fois signifie alors répéter 3 fois) ; ou bien échouer, au prétexte qu'on ne peut pas répéter un nombre flottant de fois une liste d'ordres (répéter 3.1 fois n'a pas de sens). C'est cette dernière solution que nous adoptons. Nous définissons donc une fonction de conversion d'un nombre en une valeur entière qui échoue si son argument est flottant.

```
# let valeur_entière = function
    | Entier i -> i
    | Flottant f -> failwith "entier attendu";;
valeur_entière : nombre -> int = <fun>
```

Voyons maintenant le code de la fonction qui exécute les ordres. Nous détaillerons ensuite le code qui correspond aux ordres nouveaux.

```
# exception Fin_de_procédure;;
L'exception Fin_de_procédure est définie.
# let rec exécute_ordre env = function
    | Av e -> avance (flottant (valeur_expr env e))
    | Re e -> avance (-. (flottant (valeur_expr env e)))
    | Tg a -> tourne (flottant (valeur_expr env a))
```

```
    | Td a -> tourne (-. (flottant (valeur_expr env a)))
    | Lc -> fixe_crayon true
    | Bc -> fixe_crayon false
    | Ve -> vide_écran ()
    | Rep (n, l) ->
        for i = 1 to valeur_entière (valeur_expr env n)
        do do_list (exécute_ordre env) l done
    | Si (e1, e2, alors, sinon) ->
        if compare_nombres (valeur_expr env e1, valeur_expr env e2)
        then do_list (exécute_ordre env) alors
        else do_list (exécute_ordre env) sinon
    | Stop -> raise Fin_de_procédure
    | Exécute (nom_de_procédure, args) ->
        let définition = définition_de nom_de_procédure in
        let variables = définition.paramètres
        and corps = définition.corps in
        let rec augmente_env = function
          | [], [] -> env
          | variable::vars, expr::exprs ->
             (variable, valeur_expr env expr) ::
             augmente_env (vars, exprs)
          | _ ->
             failwith ("mauvais nombre d'arguments pour "
                       ^ nom_de_procédure) in
        let env_pour_corps = augmente_env (variables, args) in
        try  do_list (exécute_ordre env_pour_corps) corps
        with Fin_de_procédure -> ();;
exécute_ordre : (string * nombre) list -> ordre -> unit = <fun>
```

L'ordre `Si` est très simple : si la comparaison des deux expressions renvoie vrai, on exécute la partie `alors` et dans le cas contraire on exécute la partie `sinon`. Remarquez que la fonction `exécute_ordre` est appliquée partiellement à l'environnement courant : c'est la fonction ainsi obtenue qui est appliquée à tous les ordres de la liste choisie (`do_list (exécute_ordre env) alors`).

L'ordre `Stop` est exécuté en déclenchant l'exception `Fin_de_procédure` qui interrompt donc brutalement l'exécution de la liste d'ordres constituant le corps d'une procédure. Cette exception est évidemment surveillée par l'exécution du corps de toute procédure ; si elle survient, elle est alors rattrapée et la procédure est considérée comme terminée : c'est le `try ... with` qui apparaît dans la clause concernant `Exécute`.

L'évaluation d'un ordre `Exécute` consiste d'abord à obtenir la définition de la procédure, ses variables et son corps. Puis on calcule l'environnement dans lequel le corps de la procédure doit être évalué ; c'est l'environnement `env_pour_corps`. Il est obtenu en liant les paramètres de la procédure aux valeurs des arguments avec lesquels la procédure a été appelée. La fonction locale `augmente_env` parcourt donc simultanément la liste des paramètres de la procédure et la liste des arguments. Si ces listes sont vides (procédure sans paramètre ou liste de paramètres complètement traitée), le nouvel environnement est l'environnement courant `env`. Sinon, il suffit de calculer la liaison du premier paramètre de la procédure, `variable`, à la valeur du premier argument, l'expression `expr`. On ajoute donc la paire `(variable, valeur_expr env`

`expr)` à la liste des autres liaisons qu'on obtient en appelant récursivement la fonction `augmente_env` sur le reste des paramètres et le reste des valeurs des arguments, les expressions `exprs`. Évidemment, si la liste des paramètres et la liste des expressions ne s'épuisent pas en même temps, c'est qu'il y a une erreur sur le nombre d'arguments fournis lors de l'appel de la procédure ; on échoue alors avec un message d'erreur. Il ne reste plus ensuite qu'à exécuter la liste des ordres du corps de la procédure dans ce nouvel environnement, en surveillant le déclenchement de l'exception `Stop`. Remarquez encore une fois l'application partielle de la fonction `exécute_ordre`, non pas à l'environnement courant, mais à celui adéquat à l'exécution du corps de la procédure : `do_list (exécute_ordre env_pour_corps) corps`.

L'évaluation des programmes

Les phrases de notre mini-Logo sont soit des définitions de procédures, soit des ordres. Un programme est une suite de phrases.

```
# type phrase_logo =
    | Pour of string * procédure
    | Ordre of ordre;;
Le type phrase_logo est défini.
# type programme_logo = Programme of phrase_logo list;;
Le type programme_logo est défini.
```

On exécute un ordre en appelant la fonction `exécute_ordre` dans un environnement initialement vide, tandis qu'une définition se contente d'appeler la fonction `définit_procédure`.

```
# let rec exécute_phrase = function
    | Ordre ord -> exécute_ordre [] ord
    | Pour (nom, proc as liaison) -> définit_procédure liaison
  and exécute_programme = function
    | Programme phs -> do_list exécute_phrase phs;;
exécute_phrase : phrase_logo -> unit = <fun>
exécute_programme : programme_logo -> unit = <fun>
```

L'analyseur syntaxique

L'analyseur syntaxique est très semblable à celui de la section 9.6 ; il est seulement un peu plus long. Nous avons regroupé les clauses analogues, dues aux mots clés synonymes, à l'aide de filtres « ou », par exemple `[< '(Mot "baisse_crayon" | Mot "bc") >] -> Bc`.

Remarquez aussi que les expressions « alors » et « sinon » de l'alternative `si` sont des listes d'ordres et que la condition est forcément la comparaison avec `>=` de deux expressions. Un petit détail encore : nous avons ajouté la possibilité de taper directement des nombres négatifs dans la fonction `nombre`. En effet, si le nombre commence par un signe `-`, nous rendons en résultat son opposé en appelant la primitive Caml correspondant au type du nombre, qui est simplement déterminé par un filtrage explicite.

```
# let rec analyse_programme = function
    | [< analyse_phrase ph; analyse_programme p >] -> ph :: p
    | [< 'Symbole '.' >] -> []
```

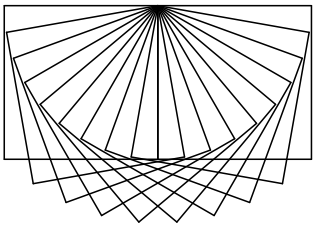
```
  | [< >] -> []

and analyse_phrase = function
  | [< 'Mot "pour"; 'Mot s; paramètres variables;
       suite_d'ordres ordres; 'Symbole '.' >] ->
     Pour (s, {paramètres = variables; corps = ordres})
  | [< ordre ord >] -> Ordre ord

and paramètres = function
  | [< 'Symbole ':'; 'Mot s; paramètres l >] -> s::l
  | [< >] -> []

and ordre = function
  | [< '(Mot "avance" | Mot "av"); expression e >] -> Av e
  | [< '(Mot "recule" | Mot "re"); expression e >] -> Re e
  | [< '(Mot "droite" | Mot "td"); expression e >] -> Td e
  | [< '(Mot "gauche" | Mot "tg"); expression e >] -> Tg e
  | [< '(Mot "baisse_crayon" | Mot "bc") >] -> Bc
  | [< '(Mot "lève_crayon" | Mot "lc") >] -> Lc
  | [< '(Mot "vide_écran" | Mot "ve") >] -> Ve
  | [< 'Mot "stop" >] -> Stop
  | [< 'Mot "si";
       expression e1; 'Symbole '>';  'Symbole '='; expression e2;
       liste_d'ordres alors;
       liste_d'ordres sinon >] -> Si (e1, e2, alors, sinon)
  | [< '(Mot "répète" | Mot "rep");
       expression e; liste_d'ordres l >] -> Rep (e,l)
  | [< 'Mot f; liste_d'expressions exprs >] -> Exécute (f, exprs)

and liste_d'ordres = function
  | [< 'Symbole '['; suite_d'ordres l; 'Symbole ']' >] -> l
and suite_d'ordres = function
  | [< ordre ord; suite_d'ordres l >] -> ord::l
  | [< >] -> []

and nombre = function
  | [< 'Symbole '-'; nombre n >] ->
    begin match n with
    | Entier i -> Entier (-i)
    | Flottant f -> Flottant (-. f)
    end
  | [< 'Constante_entière i >] -> Entier i
  | [< 'Constante_flottante f >] -> Flottant f

and expression_simple = function
  | [< nombre n >] -> Constante n
  | [< 'Symbole ':'; 'Mot var >] -> Variable var
  | [< 'Symbole '('; expression e; 'Symbole ')' >] -> e

and expression = function
  | [< expression_simple e; (reste_de_l'expression e) e' >] -> e'
```

```
  and reste_de_l'expression e = function
    | [< 'Symbole '+'; expression e2 >] -> Somme (e, e2)
    | [< 'Symbole '*'; expression e2 >] -> Produit (e, e2)
    | [< 'Symbole '-'; expression e2 >] -> Différence (e, e2)
    | [< 'Symbole '/'; expression e2 >] -> Quotient (e, e2)
    | [< >] -> e

  and liste_d'expressions = function
    | [< expression exp; liste_d'expressions l >] -> exp::l
    | [< >] -> [];;
```

La fonction principale, `logo`, combine analyse lexicale, analyse syntaxique et évaluation pour exécuter un programme lu dans une chaîne de caractères.

```
# let logo chaîne =
      do_list exécute_phrase
       (analyse_programme
         (analyseur_lexical (stream_of_string chaîne)));;
logo : string -> unit = <fun>
```

Voici en guise d'exemple les définitions successives de deux procédures, suivies d'une suite d'ordres :



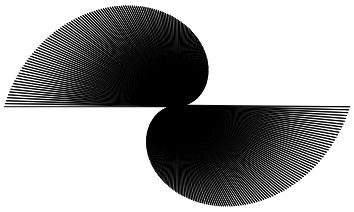
```
# logo "pour carré :c
         répète 4 [av :c td 90].
        pour multi_carré :c :n
         répète :n [carré :c td 10].
        ve multi_carré 80 10 .";;
- : unit = ()
```

Remarquez que nous devons écrire `10 .` avec un blanc entre le zéro et le point, car sinon l'analyseur lexical croirait avoir affaire à un nombre flottant.

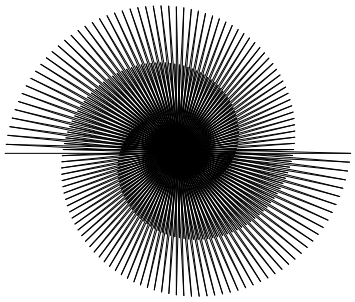
Un peu plus difficile maintenant : une procédure récursive `spirale`, qui s'arrête lorsqu'elle a été appelée `:n` fois et qui fait avancer le crayon de la distance `:d` en tournant de l'angle `:a` à chacune de ses invocations. La procédure s'appelle récursivement avec une distance augmentée de l'argument `:i`, qui est donc l'incrément ajouté à la distance que parcourra le crayon à la prochaine étape.

```
# logo "pour spirale :d :a :i :n
         si :n >= 0
          [av :d td :a spirale (:d + :i) :a :i (:n - 1)]
          [stop].";;
- : unit = ()
```

```
# logo "ve spirale
        0 179.5 0.5 360 .";;
- : unit = ()
```

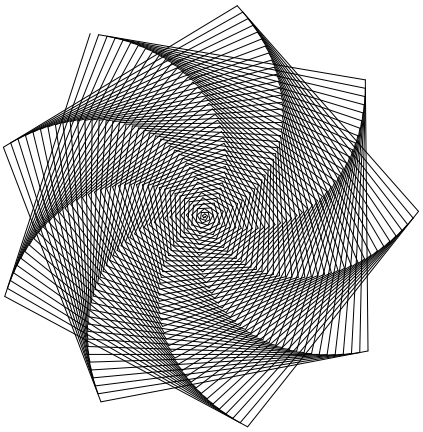


```
# logo "ve spirale
        0 178.5 0.5 360 .";;
- : unit = ()
```

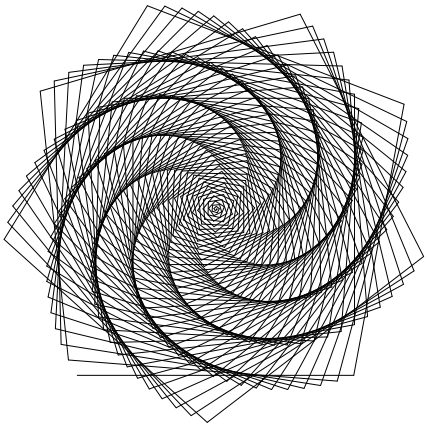


Cette fonction est étonnamment sensible à la valeur de ses paramètres de départ. Les deux dessins ci-dessus correspondent à une variation d'un degré de l'angle, toutes choses restant égales par ailleurs. De même, voyez la différence entre les deux dessins suivants, où l'angle de départ n'a été modifié que de 0.3 degrés.

```
# logo "ve spirale
        0 79.8 0.4 360 .";;
- : unit = ()
```



```
# logo "ve spirale
        0 79.5 0.4 360 .";;
- : unit = ()
```

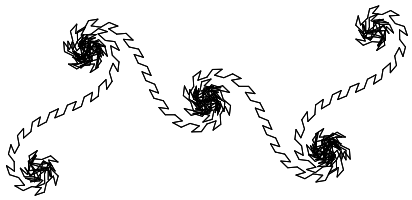


Nous écrivons maintenant une procédure analogue mais qui incrémente l'angle de visée au lieu de la distance de déplacement.

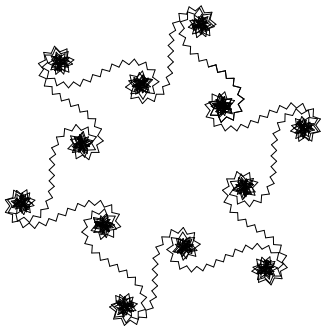
```
# logo "pour spirala :d :a :i :n
         si :n >= 0
          [av :d td :a spirala :d (:a + :i) :i (:n - 1)]
          [stop].";;
- : unit = ()
```

On obtient encore une fois des figures très diverses.

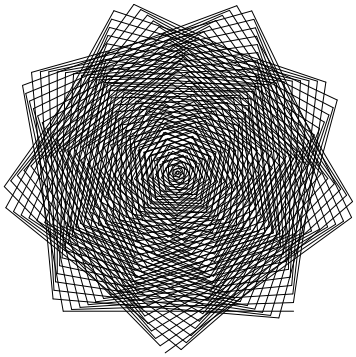
```
# logo "ve spirala
        5 0 89.5 1440 .";;
- : unit = ()
```



```
# logo "ve spirala
        4 0.5 181.5 1500 .";;
- : unit = ()
```



Au fait, `spirale` et `spirala` sont des procédures récursives. Mais qu'avons-nous fait pour que les procédures de notre mini-Logo puissent être récursives ? Comment se fait-il que nous soyons capables de les exécuter ? Nous avons vu qu'en Caml cela n'allait pas de soi puisqu'il nous avait fallu pour cela utiliser le mot-clé `rec`. Examinez les fonctions d'évaluation des ordres et de définition des procédures : vous remarquerez qu'une procédure est toujours définie sans examen de son corps. Rien ne l'empêche donc de faire référence à une autre procédure *pas encore* définie, en particulier elle peut faire référence à elle-même. Lors de l'exécution, il n'y a pas de problème non plus : on va rechercher la définition de la procédure autant de fois que nécessaire et cette recherche réussit toujours puisque la procédure est définie ... Puisque la récursivité est automatique en mini-Logo vous devinez que, à la différence de Caml, mini-Logo n'obéit pas à la règle de portée statique. En effet lors de l'exécution du corps d'une procédure, quand on rencontre un identificateur, on interroge l'environnement courant : on va donc trouver la valeur actuellement liée au nom. Cette liaison a été établie au cours de l'exécution du programme, elle n'est pas celle qu'on obtiendrait en examinant le texte du programme. Ce type de portée des identificateurs se nomme la portée *dynamique*. Il est donc légal en mini-Logo d'utiliser une procédure avant de l'avoir définie ...



10
Programmes indépendants et modules

Où l'on apprend à diviser pour régner sur ses programmes.

JUSQU'ICI, nous avons utilisé Caml d'une manière entièrement interactive : l'utilisateur entre une phrase, le système l'exécute, l'utilisateur entre une deuxième phrase, et ainsi de suite. C'est très pratique pour apprendre le langage et expérimenter avec des fonctions, mais malcommode pour construire de véritables programmes. Dans ce chapitre, nous allons voir d'autres modes d'utilisation de Caml, qui évitent d'entrer sans cesse les mêmes définitions ; qui autorisent la constructions de programmes indépendants, utilisables sans connaissance préalable de Caml ; et enfin, qui procurent un moyen de structurer des gros programmes en modules.

10.1 Chargement de fichiers

Si vous avez eu la patience de taper un des gros exemples des chapitres précédents, vous vous êtes certainement demandé comment sauvegarder tout ce travail, afin de pouvoir réutiliser ultérieurement les fonctions que vous avez péniblement entrées. Malheureusement pour vous, c'est impossible dans le système Caml Light. En revanche, vous auriez pu écrire les exemples sous un éditeur de textes, enregistrer le texte des programmes dans un fichier, puis charger ce fichier dans le système interactif. Caml Light fournit pour ce faire la fonction `include` : appliquée à un nom de fichier `include` se charge de lire les phrases Caml contenues dans le fichier et de les exécuter tout comme si vous les aviez tapées à la main. Par exemple, supposons qu'on ait mis dans le fichier `facto.ml` les lignes suivantes :

```
let rec fact n =
  if n <= 1 then 1 else n * fact (n - 1);;
fact 10;;
```

On lance alors le système Caml Light, puis on entre :

```
# include "facto.ml";;
fact : int -> int = <fun>
```

```
- : int = 3628800
- : unit = ()
```

Comme on le voit sur les réponses du système, `fact` est définie, puis `fact 10` évaluée, et le résultat est affiché. Notez que le nom de fichier argument de `include` doit impérativement être mis entre guillemets `" ... "` : c'est en fait une chaîne de caractères comme les autres. De plus, le nom du fichier doit se terminer par l'extension `.ml`. Si on donne à `include` un nom de fichier qui ne se termine pas par `.ml`, comme dans `include "facto"`, le système ajoute `.ml` de lui-même et charge donc `facto.ml`.

La méthode la plus pratique pour construire interactivement des programmes Caml Light est donc de faire tourner en même temps un éditeur de textes et le système Caml Light, dans deux fenêtres différentes : on écrit son code sous l'éditeur, on l'enregistre, on passe à Caml Light, on charge le fichier avec `include`, on lit les messages d'erreur, on repasse sous l'éditeur, on corrige les erreurs, on repasse à Caml Light, on recharge, etc.

10.2 Programmes indépendants

Supposons que l'on veuille utiliser régulièrement le système Logo présenté dans les deux chapitres précédents. On met donc dans le fichier `logo.ml` toutes les définitions nécessaires, du type `état` du chapitre 8 à la fonction `logo` du chapitre 9. Pour travailler en Logo, il faut alors lancer Caml Light, puis entrer :

```
include "logo";;
logo "une phrase Logo";;
logo "une autre phrase Logo";;
  ⋮
quit ();;
```

Cela vaut mieux certes que de réécrire tout le code du système Logo, mais ce n'est pas encore idéal : il faut expliquer aux utilisateurs que chaque phrase Logo doit être précédée de `logo "` et suivie de `";;`, ce qui n'a rien de naturel s'ils ne connaissent pas Caml. Il serait bien meilleur d'avoir un programme `logo` qu'on puisse lancer comme n'importe quel autre programme de la machine et sous lequel on puisse entrer directement les phrases Logo à exécuter.

Pour ce faire, il faut abandonner le système Caml Light interactif et utiliser le compilateur indépendant. Ce dernier transforme un fichier contenant des phrases Caml Light en un programme directement exécutable. Supposons que le fichier `hello.ml` contienne les trois phrases :

```
let message = "Bonjour tout le monde!";;
print_string message;;
print_newline ();;
```

Pour compiler ce fichier, il faut lancer le compilateur Caml Light à partir de l'interprète de commandes de votre machine. Rappelons que l'interprète de commandes est le programme système qui permet de lancer d'autres programmes en tapant des lignes de commandes. C'est par exemple le programme « Invite de commande » de MS Windows, ou l'un des nombreux « shells » du système Unix. Sur le Macintosh, le système standard ne fournit pas d'interprète de commandes. Le compilateur indépendant de Caml Light

tourne donc sous un interprète de commandes appelé MPW (Macintosh Programmer's Workshop), et disponible gratuitement auprès d'Apple. Les lecteurs qui disposent d'un Macintosh mais qui n'ont pas MPW ne peuvent pas utiliser le compilateur indépendant ; la section 10.6 explique comment procéder en utilisant seulement le système interactif.

Nous supposons que l'interprète de commandes a le symbole `$` pour signe d'invite. Les lignes qui commencent par `$` consistent donc en des commandes entrées par l'utilisateur. Les autres lignes sont les messages qu'affichent les commandes pendant qu'elles s'exécutent.

On compile le fichier `hello.ml` à l'aide de la commande

```
$ camlc -o hello hello.ml
```

Cette commande signifie « mettre dans le fichier `hello` le code compilé pour les phrases Caml contenues dans le fichier `hello.ml` ». S'il n'y a pas d'erreurs, elle se déroule sans rien afficher. Pour exécuter le code produit, il faut lancer la commande :

```
$ camlrun hello
Bonjour tout le monde!
```

Le programme a exécuté les trois phrases qui le composent, dans l'ordre, puis a rendu la main. Rien d'autre ne s'affiche : contrairement à ce qui se passe en utilisation interactive, le système n'écrit pas de lui-même le type et la valeur de chaque phrase. Dans un programme indépendant, si l'on veut faire afficher un résultat, il faut le faire explicitement en appelant une fonction d'impression comme `print_string`.

Les différences entre l'utilisation interactive et la construction de programmes indépendants sont faibles : les deux modes effectuent presque les mêmes opérations, mais dans un ordre différent. Voici ce qui se passe quand on charge un fichier interactivement avec `include` (à gauche), comparé avec ce qui se passe quand on compile ce fichier puis qu'on l'exécute (à droite) :

`# include "fichier.ml";;`
lecture de la phrase 1
typage et compilation de la phrase 1
exécution du code de la phrase 1
impression des résultats
lecture de la phrase 2
typage et compilation de la phrase 2
exécution du code de la phrase 2
impression des résultats
...

`$ camlc -o fichier fichier.ml`
lecture de la phrase 1
typage et compilation de la phrase 1
sauvegarde du code de la phrase 1
lecture de la phrase 2
typage et compilation de la phrase 2
sauvegarde du code de la phrase 2
...

`$ camlrun fichier`
exécution du code de la phrase 1
exécution du code de la phrase 2
...

10.3 Entrées-sorties de base

Reprenons nos tentatives de réalisation d'un système Logo indépendant de Caml Light. Il ne suffit pas de compiler avec `camlc` le fichier `logo.ml` contenant toutes les définitions du type `état` du chapitre 8 à la fonction `logo` du chapitre 9. Ce fichier ne

contient que des définitions de fonctions ; l'exécution du code compilé va donc exécuter les définitions de fonction, qui n'ont pas d'effet visible, puis rendre la main. Bref, notre Logo ne fera rien du tout tant qu'on n'y aura pas ajouté du code pour lire des commandes au clavier et les envoyer à la fonction d'exécution.

Voici un aperçu des fonctions d'entrée-sortie de la bibliothèque Caml Light. Pour afficher sur l'écran, on dispose des fonctions suivantes :

`print_string` s	affiche la chaîne s
`print_char` c	affiche le caractère c
`print_int` n	affiche l'entier n
`print_float` f	affiche le nombre flottant f
`print_newline ()`	affiche un retour à la ligne

Pour lire depuis le clavier, on a `read_line ()`, qui lit une ligne au clavier et renvoie la chaîne de caractères correspondante.

Pour lire et écrire sur des fichiers, au lieu d'utiliser l'écran et le clavier, il faut utiliser des canaux d'entrée (type `in_channel`) ou de sortie (type `out_channel`). L'ouverture d'un fichier en écriture, par la fonction `open_out`, renvoie un canal de sortie sur le fichier indiqué.

```
# let canal_sortie = open_out "essai.tmp";;
canal_sortie : out_channel = <abstr>
```

Pour écrire sur un canal de sortie, on dispose des fonctions `output_string` et `output_char`.

```
# output_string canal_sortie "Une ligne de texte\n";;
- : unit = ()
```

Enfin, on ferme le canal de sortie avec `close_out`.

```
# close_out canal_sortie;;
- : unit = ()
```

La lecture est très symétrique : ouverture d'un canal d'entrée avec `open_in`, lecture ligne à ligne par `input_line` ou caractère par caractère avec `input_char`, et fermeture par `close_in`.

```
# let canal_entrée = open_in "essai.tmp";;
canal_entrée : in_channel = <abstr>
# input_char canal_entrée;;
- : char = 'U'
# input_line canal_entrée;;
- : string = "ne ligne de texte"
# input_line canal_entrée;;
Exception non rattrapée: End_of_file
```

Comme on le voit sur le dernier exemple, l'exception `End_of_file` se déclenche lorsqu'on essaye de lire après la fin du fichier.

La lecture au clavier et l'affichage à l'écran sont en fait des cas particuliers d'entrées-sorties sur fichiers. Le système fournit en effet trois canaux prédéfinis :

Identificateur	Nom savant	Relié à ...
`std_in`	entrée standard du programme	clavier
`std_out`	sortie standard du programme	écran
`std_err`	sortie d'erreur du programme	écran

Par exemple, `print_string` *s* est exactement équivalent à `output_string std_out` *s* et de même `read_line ()` est synonyme de `input_line std_in`.

Une autre manière de lire un fichier caractère par caractère est de passer par l'intermédiaire d'un flux. La fonction `stream_of_channel` renvoie le flux des caractères lus depuis un canal d'entrée. En particulier, `stream_of_channel std_in` est le flux des caractères tapés au clavier. C'est cette fonction qui va nous permettre d'appliquer des analyseurs lexicaux non plus à des chaînes de caractères, mais directement à des fichiers. Voici par exemple la boucle d'interaction qu'il faut ajouter à la fin du fichier `logo.ml` pour obtenir enfin un système Logo indépendant.

```
let flux_d'entrée = stream_of_channel std_in in
let flux_lexèmes = analyseur_lexical flux_d'entrée in
while true do
  print_string "? "; flush std_out;
  try exécute_programme(analyse_programme flux_lexèmes) with
  | Parse_error ->
      print_string "Erreur de syntaxe"; print_newline ()
  | Failure s ->
      print_string ("Erreur à l'exécution: " ^ s); print_newline ()
done;;
```

Le `flush std_out` qui suit l'affichage du signe d'invite sert à garantir que l'invite est bien affichée à l'écran avant que l'on ne commence à lire la prochaine phrase. En effet, les écritures sur des canaux sont « tamponnées » (*buffered*, en anglais) pour plus d'efficacité ; autrement dit, le système accumule en mémoire un certain nombre d'ordres d'écriture et les effectue en bloc plus tard. En conséquence, évaluer `print_string "? "` ne suffit pas pour assurer qu'un point d'interrogation apparaît sur l'écran ; il faut en plus « vider » (*to flush*, en anglais) explicitement les écritures en attente. C'est le rôle de la fonction prédéfinie `flush` ; d'autres fonctions, comme `close_out` ou `print_newline`, vident aussi le canal de sortie.

Nous pouvons maintenant compiler `logo.ml` et exécuter le code produit.

```
$ camlc -o logo logo.ml
$ camlrun logo
```

Au lancement de `logo`, toutes les définitions de fonctions sont évaluées en séquence, sans effets visibles, puis la phrase ci-dessus est exécutée. La boucle infinie `while true do` ... affiche un signe d'invite, puis lit une phrase sur le flux des caractères entrés au clavier. Si le premier mot est `fin`, on sort du programme en rendant immédiatement la main à l'aide de la fonction prédéfinie `exit`. Sinon, on exécute la phrase lue et on refait un tour de boucle pour lire la suivante.

10.4 Programmes en plusieurs modules

Plutôt que de mettre tout le texte d'un programme dans un seul fichier, il est préférable de le découper en plusieurs petits fichiers, que l'on compile un par un. Non seulement l'édition et la recompilation sont facilitées, mais surtout on s'autorise alors la réutilisation de certains morceaux du programme dans d'autres programmes. Par exemple, les fonctions sur le crayon électronique (`avance`, ...) sont susceptibles d'être

utilisées dans bien d'autres programmes que notre système mini-Logo. On appelle *modules* ces morceaux de programme suffisamment autonomes pour être éventuellement réutilisés plus tard et *programmation modulaire* le style de programmation consistant à découper systématiquement les programmes en modules.

Nous allons donc découper le mini-Logo en cinq modules :

`crayon`	le crayon électronique : fonctions `avance`, `tourne`, ...
`langage`	le langage de commandes : type `ordre`, fonction `exécute_ordre`
`alex`	l'analyseur lexical : type `lexème`, fonction `lire_lexème`, ...
`asynt`	l'analyseur syntaxique : fonction `analyse_programme`, ...
`logo`	le programme principal : la boucle d'interaction.

À chaque module correspond un fichier source, qui a le même nom que le module, avec l'extension `.ml`. Par exemple, les fonctions du module `crayon` sont définies dans le module `crayon.ml`. Le contenu de ces fichiers est résumé figure 10.1.

Noms extérieurs

La figure 10.1 montre que nous avons ajouté à chaque module des lignes de la forme `#open` plus un nom de module. Ces lignes indiquent d'où proviennent les noms extérieurs qu'on utilise dans le fichier sans les y avoir définis. Grâce à ces indications, le compilateur sait où aller chercher le type et le code compilé de ces noms extérieurs.

Il y a deux manières de faire référence à un identificateur extérieur. L'une est d'utiliser des noms « qualifiés », de la forme : nom du module d'origine, suivi de deux caractères `_` (souligné), suivi du nom de l'identificateur. Ainsi, `asynt__analyse_programme` signifie « l'identificateur `analyse_programme` défini dans le module `asynt` ».

L'autre manière d'accéder à des identificateurs extérieurs est d'introduire des directives `#open "`*module*`"`. Cette directive indique au compilateur qu'on veut « ouvrir » (*to open*, en anglais) le module donné en argument. Plus précisément, cette directive dit que si l'on rencontre un identificateur non qualifié qui n'est pas défini dans le fichier en cours de compilation, il faut le chercher dans le module argument de `#open`. Par exemple, dans le fichier `asynt.ml`, après la ligne

```
#open "langage";;
```

on peut faire référence au type `ordre` et à ses constructeurs par des identificateurs simples (`Av`, `Re`, ...). Sans le `#open`, il aurait fallu utiliser des noms qualifiés (`langage__Av`, ...). Plusieurs directives `#open` dans un fichier donnent ainsi au compilateur une liste de modules où aller chercher les identificateurs externes.

Le choix entre ces deux manières de faire référence à un nom extérieur est une pure question de style : l'emploi de `#open` donne des programmes plus compacts et permet de renommer les modules plus facilement ; l'emploi de noms qualifiés montre plus clairement la structure modulaire du programme.

La bibliothèque de modules du système

Il n'y a pas que les programmes de l'utilisateur à être découpés en modules : la bibliothèque de fonctions prédéfinies du système Caml Light se présente elle aussi sous la

Fichier `crayon.ml`

```
#open "graphics";;

let round x = ... ;;
type état = ... ;;
let crayon = ... ;;
let avance d = ... ;;
let pi_sur_180 = ... ;;
let tourne angle = ... ;;
let avance d = ... ;;
let couleur_du_tracé = ... ;;
let couleur_du_fond = ... ;;
let zéro_x = ... ;;
let zéro_y = ... ;;
let vide_écran () = ... ;;
```

Fichier `langage.ml`

```
#open "crayon";;

type nombre = ...;;
let flottant = ... ;;
type expression = ... ;;
let ajoute_nombres = ... ;;
let soustrait_nombres = ... ;;
let multiplie_nombres = ... ;;
let divise_nombres = ... ;;
let compare_nombres = ... ;;
let rec valeur_expr env = ... ;;
type ordre = ... ;;
type procédure = ...;;
let procédures_définies = ... ;;
let définit_procédure = ...
and définition_de = ... ;;
let valeur_entière = ... ;;
let rec exécute_ordre env = ...;;
type phrase_logo = ... ;;
type programme_logo = ... ;;
let exécute_phrase = ...
let exécute_programme = ... ;;
```

Fichier `alex.ml`

```
type lexème = ... ;;
let rec saute_blancs = ... ;;
let rec lire_entier = ... ;;
let rec lire_décimales = ... ;;
let rec lire_mot = ... ;;
let lire_lexème = ... ;;
let rec analyseur_lexical = ...;;
```

Fichier `asynt.ml`

```
#open "langage";;
#open "alex";;

let rec analyse_programme = ...
and analyse_phrase = ...
and paramètres = ...
and ordre = ...
and liste_d'ordres = ...
and suite_d'ordres = ...
and nombre = ...
and expression_simple = ...
and expression = ...
and reste_de_l'expression = ...
and liste_d'expressions = ... ;;
```

Fichier `logo.ml`

```
#open "langage";;
#open "alex";;
#open "asynt";;

let flux_d'entrée = ... in
let flux_lexèmes = ... in
while true do
  ...
done;;
```

Figure 10.1: Découpage en modules de l'interprète mini-Logo

forme d'un certain nombre de modules. Par exemple, la fonction `sub_string` provient du module de bibliothèque `string` ; de même, des opérateurs comme `+` et `+.` ne sont pas entièrement prédéfinis dans le système, mais proviennent de modules de bibliothèque (les modules `int` et `float`, respectivement). Certains de ces modules de bibliothèque (comme `int`, `float` et `string`) sont implicitement « ouverts » au lancement du compilateur. Tout se passe comme si on avait mis au début de tous les fichiers :

```
#open "int";;   #open "float";;  #open "string";;
```

C'est ce qui explique qu'on référence directement `sub_string` dans n'importe quel programme, sans mettre au préalable `#open "string"` ni devoir utiliser la forme complètement qualifiée `string__sub_string`.

D'autres modules de bibliothèque, d'un emploi moins fréquent, ne sont pas « ouverts » automatiquement au début de chaque compilation. C'est le cas par exemple du module `graphics` fournissant les commandes graphiques de base. Il faut donc mettre `#open "graphics"` au début du fichier `crayon.ml`, qui fait référence à ces commandes graphiques.

Compilation séparée

Les modules composant un programme se compilent un par un à l'aide de la commande `camlc -c`.

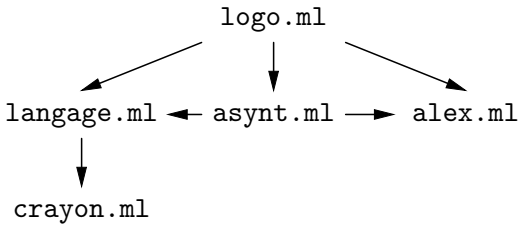
```
$ camlc -c crayon.ml
$ camlc -c langage.ml
$ camlc -c alex.ml
$ camlc -c asynt.ml
$ camlc -c logo.ml
```

L'option `-c` indique au compilateur qu'il ne faut pas essayer de produire un fichier de code exécutable. En d'autres termes, cette option prévient le compilateur que le fichier donné en argument n'est pas un programme complet, mais seulement un morceau de programme. L'exécution de la commande `camlc -c crayon.ml` produit deux fichiers :

- le fichier `crayon.zo`, qu'on appelle « fichier de code objet » ; il contient du code compilé pas encore exécutable, car faisant référence à des identificateurs extérieurs ;
- le fichier `crayon.zi`, qu'on appelle « fichier d'interface compilée » ; il contient des informations de typage sur les objets déclarés dans le module `crayon` : types des identificateurs définis, noms des types concrets déclarés avec leurs constructeurs, etc.

Le fichier d'interface compilée `crayon.zi` sert pour la compilation des modules qui utilisent le module `crayon` : quand on compile un module contenant `#open "crayon"` ou un nom qualifié de la forme `crayon__`..., le compilateur lit le fichier `crayon.zi` et y trouve toutes les informations de typage dont il a besoin.

Ce comportement introduit une contrainte sur l'ordre dans lequel on compile les modules : lorsqu'on compile un module, il faut avoir compilé au préalable tous les modules qu'il utilise, ayant ainsi produit tous les fichiers `.zi` nécessaires à sa compilation. Dans le cas du mini-Logo, ces contraintes se résument par le schéma suivant (une flèche de A vers B signifie que A utilise B et donc que B doit être compilé avant A).



La séquence de compilation donnée ci-dessus vérifie toutes les contraintes. On a aussi la liberté de compiler `alex.ml` plus tôt, avant `crayon` ou `langage`.

Édition de liens

Lorsque tous les modules composant le programme ont été compilés, il faut lier ensemble leurs fichiers de code objet, obtenant ainsi un fichier exécutable par `camlrun`. Cette opération s'appelle l'édition de liens et s'effectue en appelant `camlc` avec la liste des fichiers en `.zo` à lier ensemble.

```
$ camlc -o logo crayon.zo langage.zo alex.zo asynt.zo logo.zo
```

Comme pour un programme mono-fichier, l'option `-o` sert à donner le nom du fichier exécutable à produire. L'ordre des fichiers `.zo` sur la ligne a son importance : il doit respecter la même contrainte que pour l'ordre des compilations, à savoir qu'un module doit apparaître avant les modules qui l'utilisent.

10.5 Interfaces de modules

Souvent, un module contient des définitions à usage interne, qui ne sont pas censées être employées à l'extérieur du module. Dans le module `crayon`, par exemple, l'enregistrement à champs mutables qui contient l'état courant de la tortue n'est pas censé être modifié directement par les utilisateurs de ce module ; les utilisateurs sont supposés passer par l'intermédiaire des fonctions `avance`, `tourne`, ... On peut s'imposer de respecter cette convention soi-même ; mais le système est capable de la garantir, si on lui demande explicitement de « cacher » certains des identificateurs définis par le module, ce qui les rend ainsi inaccessibles depuis l'extérieur du module.

Pour ce faire, il faut écrire une interface au module. L'interface d'un module contient des déclarations pour tous les identificateurs du module que l'on veut rendre visibles de l'extérieur ; les identificateurs définis dans le module mais non déclarés dans l'interface seront automatiquement cachés. L'interface d'un module réside dans un fichier ayant le même nom que le module mais avec l'extension `.mli`. Par opposition, le fichier avec l'extension `.ml` qui contient les définitions du module s'appelle l'implémentation du module. Par exemple, voici le fichier d'interface du module `crayon` :

```
value vide_écran: unit -> unit
  and fixe_crayon: bool -> unit
  and tourne: float -> unit
  and avance: float -> unit;;
```

Comme on le voit, les déclarations d'identificateurs sont introduites par le mot-clé `value` et consistent en le nom de l'identificateur suivi de son type. Les interfaces peuvent aussi

contenir des définitions de types et d'exceptions. Par exemple, l'interface du module `alex` rend public le type des lexèmes, en plus de la fonction d'analyse lexicale.

Fichier `alex.mli`

```
type lexème =
   | Mot of string
   | Symbole of char
   | Entier of int
   | Flottant of float;;

value analyseur_lexical: char stream -> lexème stream;;
```

On trouvera en figure 10.2 la nouvelle structure du mini-Logo, une fois qu'on a ajouté des interfaces à tous les modules (sauf le module principal `logo`, qui ne définit rien de toute façon). Remarquez que si un type est défini dans l'interface d'un module, il est automatiquement défini dans l'implémentation du module ; il ne faut donc pas recopier sa définition dans cette implémentation.

Compilation des interfaces

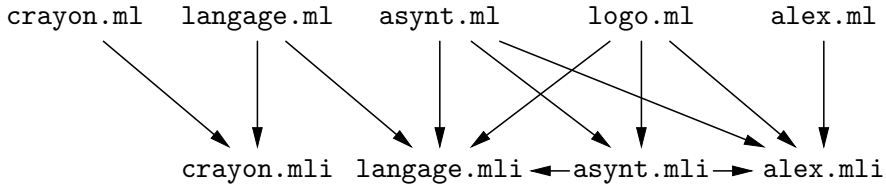
Les fichiers d'interface se compilent exactement comme les fichiers d'implémentation, par la commande `camlc -c`. Exemple :

```
$ camlc -c crayon.mli
```

L'exécution de cette commande crée un fichier `crayon.zi` contenant les déclarations de `crayon.mli` sous une forme compilée. Comme dans le cas des modules sans interface (section 10.4), le fichier `crayon.zi` est consulté par le compilateur lors de la compilation des modules qui font référence au module `crayon`. De plus, lorsqu'on compile l'implémentation `crayon.ml`,

```
$ camlc -c crayon.ml
```

le compilateur vérifie la cohérence de l'implémentation avec l'interface compilée `crayon.zi`, c'est-à-dire qu'il vérifie que tous les identificateurs déclarés dans l'interface sont bien définis dans l'implémentation et qu'ils ont bien le type annoncé dans l'interface. C'est en cela que la compilation d'un module avec interface explicite diffère de la compilation d'un module sans interface : si l'interface `.mli` existe alors le `.zi` est construit par compilation du `.mli` et la compilation de l'implémentation `.ml` consulte le `.zi` pour vérifier la cohérence ; si l'interface `.mli` n'existe pas, alors la compilation de l'implémentation `.ml` créé un `.zi` qui rend public tout ce que l'implémentation `.ml` définit. Il en découle deux contraintes sur l'ordre dans lequel on effectue les compilations : d'une part, l'interface explicite *mod*`.mli` doit être compilée avant tous les fichiers (`.ml` et `.mli`) qui font référence au module *mod* ; d'autre part, l'interface explicite *mod*`.mli` doit être compilée avant l'implémentation *mod*`.ml`. Dans le cas du mini-Logo, il en découle les contraintes suivantes :



Fichier `crayon.mli`

```
value vide_écran: unit -> unit
  and fixe_crayon: bool -> unit
  and tourne: float -> unit
  and avance: float -> unit;;
```

Fichier `crayon.ml`

```
#open "graphics";;
let round x = ... ;;
type état = ... ;;
let crayon = ... ;;
let avance d = ... ;;
let pi_sur_180 = ... ;;
let tourne angle = ... ;;
let avance d = ... ;;
let couleur_du_tracé = ... ;;
let couleur_du_fond = ... ;;
let zéro_x = ... ;;
let zéro_y = ... ;;
let vide_écran () = ... ;;
```

Fichier `langage.mli`

```
type nombre = ... ;;
type expression = ... ;;
type ordre = ... ;;
type procédure = ...;;
type phrase_logo = ... ;;
type programme_logo = ... ;;
value exécute_phrase:
        phrase_logo -> unit
  and exécute_programme:
        programme_logo -> unit;;
```

Fichier `langage.ml`

```
#open "crayon";;
let flottant = ... ;;
let ajoute_nombres = ... ;;
(* ... *)
let rec valeur_expr env = ... ;;
let procédures_définies = ... ;;
let définit_procédure = ...
and définition_de = ... ;;
let rec exécute_ordre env = ...;;
let exécute_phrase = ...
let exécute_programme = ... ;;
```

Fichier `alex.mli`

```
type lexème = ... ;;
value analyseur_lexical:
   char stream -> lexème stream;;
```

Fichier `alex.ml`

```
let rec saute_blancs = ... ;;
let rec lire_entier = ... ;;
let rec lire_décimales = ... ;;
let rec lire_mot position = ...;;
let lire_lexème = ... ;;
let rec analyseur_lexical = ...;;
```

Fichier `asynt.mli`

```
value analyse_phrase:
      alex__lexème stream ->
        langage__phrase_logo
  and analyse_programme:
      alex__lexème stream ->
        langage__programme_logo;;
```

Fichier `asynt.ml`

```
#open "langage";;
#open "alex";;
let rec analyse_programme = ...
and analyse_phrase = ...
and paramètres = ...
and ordre = ...
and liste_d'ordres = ...
and suite_d'ordres = ...
and nombre = ...
and expression_simple = ...
and expression = ...
and reste_de_l'expression = ...
and liste_d'expressions = ... ;;
```

Fichier `logo.ml`

```
#open "langage";;
#open "alex";;
#open "asynt";;
let flux_d'entrée = ... in
let flux_lexèmes = ... in
while true do
  ...
done;;
```

Figure 10.2: Ajout d'interfaces explicites aux modules de l'interprète mini-Logo

Remarquez que les fichiers d'implémentation (`.ml`) sont compilables dans n'importe quel ordre : si un module A utilise un module B, on peut très bien compiler l'implémentation de A avant l'implémentation de B ; il suffit que l'interface de B ait déjà été compilée. C'est le cas dans la séquence de compilation ci-dessous.

```
$ camlc -c langage.mli
$ camlc -c crayon.mli
$ camlc -c langage.ml
$ camlc -c alex.mli
$ camlc -c asynt.mli
$ camlc -c logo.ml
$ camlc -c asynt.ml
$ camlc -c crayon.ml
$ camlc -c alex.ml
```

On a choisi d'écrire et de compiler d'abord les implémentations de `langage` et de `logo`, qui représentent le cœur du système, et de repousser à plus tard l'écriture de `asynt.ml`, `alex.ml` et `crayon.ml`. Plus généralement, l'introduction d'interfaces explicites pour les modules permet de se libérer du style d'écriture des programmes strictement ascendant (*bottom-up*, en anglais) que nous avons utilisé jusqu'ici. Par la suite, nous utiliserons des interfaces à chaque fois que nous avons besoin d'un module de fonctions auxiliaires, dont nous préférons cependant repousser l'implémentation à plus tard.

10.6 Compilations interactives

Le compilateur indépendant n'est pas le seul moyen de compiler des fichiers : le système interactif fournit lui aussi des commandes pour compiler des fichiers (`compile`) et charger et exécuter du code compilé (`load_object`). La commande

```
# compile "monfichier.ml";;
```

compile le fichier `monfichier.ml` exactement comme le ferait l'appel `camlc -c monfichier.ml` du compilateur indépendant. La fonction `compile` accepte également des fichiers d'interface (`.mli`) en plus des fichiers d'implémentation (`.ml`). La commande

```
# load_object "monfichier.zo";;
```

charge en mémoire le code compilé qui se trouve dans le fichier `monfichier.zo` et l'exécute phrase par phrase. Les deux fonctions `compile` et `load_object` permettent donc de compiler et d'exécuter un programme écrit sous forme modulaire à partir du système interactif, ce qui est précieux pour les utilisateurs de Macintosh ne disposant pas du compilateur indépendant. Par exemple, le programme `logo` se compile et s'exécute de la manière suivante :

```
# compile "langage.mli";;
# compile "crayon.mli";;
# compile "langage.ml";;
# compile "alex.mli";;
# compile "asynt.mli";;
# compile "logo.ml";;
# compile "asynt.ml";;
# compile "crayon.ml";;
# compile "alex.ml";;
```

```
# load_object "crayon.zo";;
# load_object "langage.zo";;
# load_object "alex.zo";;
# load_object "asynt.zo";;
# load_object "logo.zo";;
```

Même lorsqu'on dispose du compilateur indépendant, charger du code compilé dans le système interactif avec la fonction `load_object` est souvent très utile pour essayer et mettre au point les fonctions d'un programme. Par exemple, pour tester l'analyseur syntaxique du mini-Logo, il suffit de lancer le système interactif et de faire :

```
# load_object "alex.zo";;
# load_object "asynt.zo";;
# asynt__analyse_phrase
    (alex__analyseur_lexical
      (stream_of_string "ve av 15 ."));;
```

L'utilisation interactive de modules compilés indépendamment est un peu délicate. En particulier, il faut charger les fichiers de code compilé dans le bon ordre (un module doit être chargé avant les modules qui l'utilisent). Ces points sont expliqués en détails dans le chapitre 4 du *Manuel de référence du langage Caml*.

11
Interfaces graphiques

Où Caml attrape des boutons et ouvre les fenêtres.

Humains et machines n'interagissent pas uniquement par l'intermédiaire d'un clavier et d'un écran de texte. Nous montrons dans ce chapitre comment doter les programmes Caml de jolies interfaces homme-machine graphiques, avec menus, boutons, barres de défilement, etc. Nous utiliserons pour cela la bibliothèque CamlTk, qui fournit une interface simple avec la boîte à outils (*toolkit*) Tk. Les programmes de ce chapitre peuvent être exécutés avec `camltktop`, une version du système interactif Caml qui intègre la bibliothèque CamlTk ; elle se lance par la commande `camllight camltktop`.

11.1 Structure d'une interface graphique

Une interface utilisateur réalisée avec CamlTk est constituée d'un certain nombre de *composants* de base (en anglais, *widgets*), tels que boutons, menus, zones d'entrée de texte, etc. Le programme crée les composants dont il a besoin pour interagir avec l'utilisateur, et les place dans une ou plusieurs fenêtres à l'écran. Il associe des fonctions Caml à certains types d'événement, tels que un clic de la souris dans un bouton, la sélection d'une entrée de menu, ou l'appui d'une touche du clavier. Finalement, il appelle la boucle d'interaction de CamlTk, qui affiche tous les composants à l'écran, gère l'interaction avec l'utilisateur, et appelle les fonctions Caml correspondant aux événements qui intéressent le programme.

Commençons par un exemple très simple : une interface réduite à un seul bouton, qui affiche le message `Bonjour!` lorsque l'utilisateur clique sur le bouton.

```
# #open "tk";;
  let fenêtre_principale = openTk () in
  let action () = print_string "Bonjour!"; print_newline () in
  let bouton =
    button__create fenêtre_principale
                   [Text "Pressez-moi"; Command action] in
  pack [bouton] [];
```

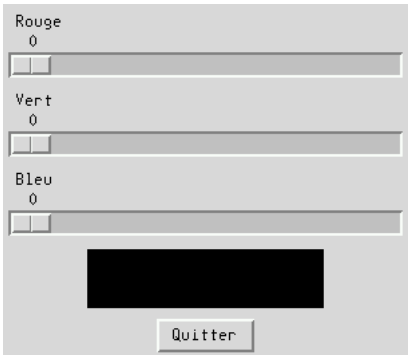
```
mainLoop ();;
```

Comme tous les programmes CamlTk, celui-ci commence par un appel à `openTk`. Cette fonction initialise la bibliothèque CamlTk, et renvoie un composant qui représente la fenêtre principale de l'application. Ensuite, le programme crée le bouton qui constitue l'essentiel de son interface, à l'aide de la fonction `button__create`. Comme toutes les fonctions de création de composants CamlTk, `button__create` prend deux arguments : un composant « père » et une liste d'options qui déterminent l'apparence et le comportement du bouton. Le composant « père » est le composant à l'intérieur duquel le bouton doit apparaître ; ici, c'est la fenêtre principale de l'application, telle que renvoyée par `openTk ()`. Quant aux options, nous en donnons ici deux : `Text "Pressez-moi"`, qui indique que le bouton doit porter l'étiquette « Pressez-moi », et `Command action`, qui associe la fonction Caml `action` aux clics sur le bouton.

Le programme appelle ensuite la fonction `pack` (« emballer », en anglais) pour effectuer le placement des composants dans les fenêtres. La fonction `pack` prend une liste de composants et une liste d'options expliquant comment placer les composants les uns par rapport aux autres (l'un au-dessus de l'autre ou l'un à côté de l'autre, avec ou sans espace entre les composants, etc.). Ici, le placement est très simple, puisqu'il n'y a qu'un seul composant dans notre interface. Enfin, la fonction `mainLoop` est appelée. Celle-ci ouvre la fenêtre à l'écran, affiche le bouton dedans, et appelle la fonction `action` à chaque fois que l'utilisateur clique sur le bouton. La fonction `mainLoop` ne rend la main que lorsque l'utilisateur a fermé la fenêtre CamlTk, ou bien interrompu le programme.

11.2 Relier des composants entre eux

Passons maintenant à un exemple plus intéressant, qui introduit deux nouveaux types de composants (les glissières et les cadres), et montre comment coupler entre eux les états de plusieurs composants. Il s'agit d'un programme permettant de choisir des couleurs à l'écran et jouant sur les intensités des trois couleurs primaires (rouge, vert, bleu). Trois glissières contrôlent ces intensités ; dès que l'utilisateur déplace l'une des glissières, la couleur correspondante est affichée dans le rectangle en bas de la fenêtre.




```
# let fenêtre_principale = openTk ();;
  let créer_glissière nom =
    scale__create fenêtre_principale
                  [Label nom; From 0.0; To 255.0;
                   Length(Centimeters 10.0); Orient Horizontal];;
  let rouge = créer_glissière "Rouge"
  and vert  = créer_glissière "Vert"
  and bleu  = créer_glissière "Bleu"
  and échantillon =
    frame__create fenêtre_principale
```

```
                  [Height(Centimeters 1.5); Width(Centimeters 6.0)]
  and quitter =
    button__create fenêtre_principale
                   [Text "Quitter"; Command closeTk];;
```

Nous commençons par créer trois glissières (*scale*) pour chacune des couleurs primaires rouge, vert et bleu. Ces glissières prennent des valeurs entre 0 et 255 (`From 0.0; To 255.0`), ont une longueur de 10 cm (`Length(Centimeters 10.0)`) et sont orientées dans le sens horizontal (`Orient Horizontal`). Pour afficher la couleur, nous créons également un cadre (*frame*), qui est une zone inactive de 6 cm sur 1,5 cm (`Height(Centimeters 1.5); Width(Centimeters 6.0)`). Enfin, le dernier composant du programme est un bouton étiqueté « Quitter » dont l'action (`closeTk`) est de fermer la fenêtre de CamlTk, terminant ainsi la boucle d'interaction `mainLoop`.

```
# let rafraîchir_couleur x =
    let r = int_of_float(scale__get rouge)
    and v = int_of_float(scale__get vert)
    and b = int_of_float(scale__get bleu) in
    let couleur = printf__sprintf "#%02x%02x%02x" r v b in
    frame__configure échantillon [Background (NamedColor couleur)];;
```

La fonction `rafraîchir_couleur` est le cœur du programme : elle change la couleur d'`échantillon` pour refléter l'état courant des trois glissières. Elle lit la valeur numérique courante des trois glissières à l'aide de `scale__get`, puis construit le nom CamlTk de la couleur correspondante. Ce nom est de la forme `#`*rrvvbb*, où *rr*, *vv* et *bb* sont les intensités de rouge, de vert et de bleu, exprimées sous forme d'un nombre hexadécimal à deux chiffres.

Le nom de la couleur est construit à l'aide de la fonction `sprintf` du module `printf`, qui est un puissant outil d'impression formatée. La fonction `sprintf` prend en argument une chaîne de caractère, le « format », et un certain nombre d'entiers ou de chaînes de caractères ; elle affiche la chaîne de caractères, en remplaçant les séquences de la forme `%` plus une lettre par le prochain argument. Par exemple, `sprintf "%d + %d" 1 2` renvoie la chaîne `"1 + 2"`. Les chiffres et la lettre suivant `%` indiquent le type de l'argument à afficher et le format d'affichage à utiliser. De nombreux formats sont disponibles. Dans l'exemple ci-dessus, nous avons utilisé `%d`, qui convertit un entier en décimal. Dans la fonction `rafraîchir_couleur`, nous utilisons `%02x`, qui convertit un entier en hexadécimal (`x`), sur deux chiffres (`2`), en complétant à gauche avec des zéros si nécessaire (`0`).

Enfin, nous changeons la couleur de fond du cadre `échantillon` à l'aide de la fonction `frame__configure`. De manière générale, toutes les options qui peuvent être spécifiées au moment où l'on crée un composant (en second argument de *composant*`__create`) peuvent aussi être spécifiées ou modifiées plus tard *via* la fonction *composant*`__configure`. Mettant immédiatement ce principe en pratique, nous utilison `scale__configure` pour associer la fonction `rafraîchir_couleur` au déplacement de chacune des glissières :

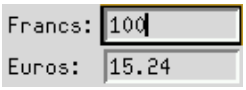
```
# scale__configure rouge [ScaleCommand rafraîchir_couleur];
  scale__configure vert [ScaleCommand rafraîchir_couleur];
  scale__configure bleu [ScaleCommand rafraîchir_couleur];
  pack [rouge; vert; bleu] [Side Side_Top];
```

```
  pack [quitter] [Side Side_Bottom];
  pack [échantillon] [Side Side_Bottom; PadY(Millimeters 2.0)];
  mainLoop ();;
```

Comme dans le premier exemple, nous plaçons finalement tous les composants à l'aide de `pack`, puis lançons l'interaction avec l'utilisateur en appelant `mainLoop ()`. Le placement se fait en trois temps : d'abord, les trois glissières en haut de la fenêtre (`Side Side_Top`) ; puis le bouton « Quitter » en bas (`Side Side_Bottom`) ; enfin, le cadre `échantillon` en bas de l'espace restant libre (`Side Side_Bottom`), et avec une marge de 2 mm en haut et en bas (`PadY(Millimeters 2.0)`) pour que ce soit plus joli.

11.3 Un convertisseur de devises

Notre prochain exemple est d'actualité : il s'agit d'une calculatrice de conversion francs-euros. Elle introduit plusieurs nouveaux types de composant : les zones d'entrée, dans lesquelles l'utilisateur peut taper et éditer un texte ; les étiquettes ; les menus déroulants. Elle illustre également le mécanisme général de *liaison* de CamlTk, qui permet d'associer une action Caml à presque n'importe quel type d'événement (appui de touche, clic de souris, mouvement de la souris, etc.).



La calculatrice se compose de deux zones d'entrée, l'une pour les francs, l'autre pour les euros. Dès que l'utilisateur modifie le montant figurant dans l'une des zones, ce montant est converti dans l'autre monnaie et affiché dans l'autre zone. Nous commençons par la fonction centrale de l'application, qui assure cette mise à jour automatique d'une zone d'entrée (le paramètre `dest`) lorsque l'autre zone (le paramètre `source`) change.

```
# let synchronise_zones source dest taux_source taux_dest =
    function infos ->
      try
        let montant_source = float_of_string (entry__get source) in
        let montant_dest =
          montant_source *. !taux_source /. !taux_dest in
        entry__delete_range dest (At 0) End;
        entry__insert dest (At 0)
                           (printf__sprintf "%.2f" montant_dest)
      with Failure _ ->
        entry__delete_range dest (At 0) End;
        entry__insert dest (At 0) "erreur";;
```

La fonction `entry__get` renvoie le texte qui figure actuellement dans un composant « zone d'entrée » (*entry*). Ce texte peut être modifié par le programme à l'aide de `entry__delete_range`, qui efface un intervalle de caractères (ici, depuis `At 0`, le premier caractère, jusqu'à `End`, le dernier caractère, effaçant ainsi tout le texte), et `entry__insert`, qui insère la chaîne donnée en argument à la position spécifiée (ici, `At 0`, c'est-à-dire le début de la zone).

```
# let fp = openTk ();;
  let ligne1 = frame__create fp [] and ligne2 = frame__create fp [];;
```

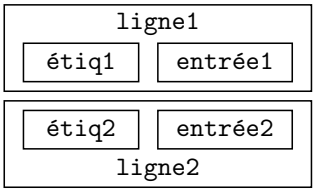
```
  let étiq1 = label__create ligne1 [Text "Francs:"]
  and entrée1 = entry__create ligne1 [TextWidth 10; Relief Sunken]
  and étiq2 = label__create ligne2 [Text "Euros:"]
  and entrée2 = entry__create ligne2 [TextWidth 10; Relief Sunken];;
```

Nous créons maintenant les quatre composants de notre application : deux zones d'entrée et deux « étiquettes » (*label*), qui sont des zones passives affichant un texte. Pour permettre l'arrangement « en carré » de ces composants (voir figure ci-dessous), nous créons également deux cadres, `ligne1` et `ligne2`, qui servent de composants père à `étiq1` et `entrée1` pour l'un, et `étiq2` et `entrée2` pour l'autre.

```
# let taux1 = ref 1.0     (* francs pour 1 franc *)
  and taux2 = ref 6.55957 (* francs pour 1 euro *);;
  bind entrée1 [[], KeyRelease]
       (BindSet([], synchronise_zones entrée1 entrée2 taux1 taux2));;
  bind entrée2 [[], KeyRelease]
       (BindSet([], synchronise_zones entrée2 entrée1 taux2 taux1));;
```

Ci-dessus, nous associons la fonction `synchronise_zones` à l'événement « relâcher une touche enfoncée » (`KeyRelease`) dans les deux zones d'entrée. La fonction `bind` gère les associations de fonctions Caml à des événements : elle prend en arguments le composant concerné, une liste d'événements et de modificateurs (ici, `[[], KeyRelease]`, signifiant la relâche d'une touche sans modificateurs particuliers), et une action à effectuer (ici, `BindSet`, signifiant la définition d'une nouvelle liaison ; on dispose aussi de `BindRemove` pour enlever toutes les liaisons, et de `BindExtend` pour rajouter une liaison). Nous avons choisi de lié l'événement `KeyRelease` plutôt que `KeyPressed` (appui d'une touche), afin que la fonction Caml soit appelée après l'insertion du caractère frappé dans la zone d'entrée, et non pas avant, ce qui produirait des résultats visuellement faux.

```
# pack [étiq1] [Side Side_Left];
  pack [entrée1] [Side Side_Right];
  pack [étiq2] [Side Side_Left];
  pack [entrée2] [Side Side_Right];
  pack [ligne1; ligne2]
       [Side Side_Top; Fill Fill_X];
  mainLoop ();;
```



Nous appelons `pack` cinq fois pour réaliser l'arrangement en carré des composants décrit par la figure ci-dessus, puis lançons la boucle d'interaction.

Choix des devises par un menu

Il n'est pas difficile d'étendre notre convertisseur à d'autres devises que le franc et l'euro. Pour ce faire, nous ajoutons deux menus déroulants permettant de choisir les devises « source » et « cible » de la conversion.

```
# let barre_de_menus =
    frame__create fp [Relief Raised; BorderWidth (Pixels 2)];;
  let bouton_source =
    menubutton__create barre_de_menus
                       [Text "Source"; UnderlinedChar 0]
  and bouton_cible =
```

```
    menubutton__create barre_de_menus
                       [Text "Cible"; UnderlinedChar 0];;
  let source = menu__create bouton_source []
  and cible = menu__create bouton_cible [];;
  menubutton__configure bouton_source [Menu source];
  menubutton__configure bouton_cible [Menu cible];
  pack [bouton_source; bouton_cible] [Side Side_Left];;
```

La barre de menus se compose d'un cadre (`barre_de_menus`), de deux « boutons à menus » (`bouton_source` et `bouton_cible`) dont l'effet est de dérouler les menus correspondants, et enfin de deux composants de type « menu » (`source` et `cible`) qui contiennent les entrées des menus. Les menus sont pour l'instant vides ; nous allons les remplir dans le code qui suit.

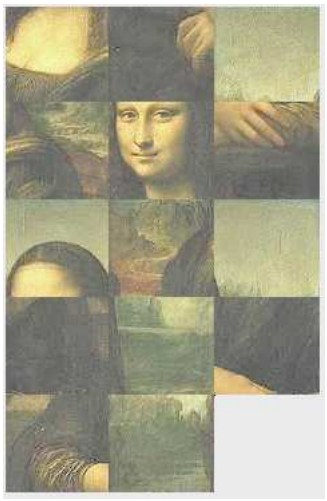
```
# let liste_de_devises =
    [ "Dollars US", 5.9389;  "Dollars canadiens", 3.933046;
      "Euros", 6.55957;  "Francs", 1.0;  "Francs belges", 0.162531;
      "Francs suisses", 4.116;  "Lires", 0.00338617;  "Livres", 9.552;
      "Marks", 3.354;  "Pesetas", 0.0394061;  "Yens", 0.05011 ];;
  do_list
    (function (nom, taux) ->
      menu__add_command source
        [Label nom;
         Command(function () ->
                   label__configure étiq1 [Text(nom ^ ":")];
                   taux1 := taux;
                   synchronise_zones entrée1 entrée2 taux1 taux2 ())])
    liste_de_devises;
  do_list
    (function (nom, taux) ->
      menu__add_command cible
        [Label nom;
         Command(function () ->
                   label__configure étiq2 [Text(nom ^ ":")];
                   taux2 := taux;
                   synchronise_zones entrée2 entrée1 taux2 taux1 ())])
    liste_de_devises;;
```

Pour chaque devise, nous ajoutons une entrée dans le menu « Source » et une entrée dans le menu « Cible ». Les actions associées à ces entrées de menus changent le nom de la devise dans l'étiquette correspondante, puis ajustent le taux de conversion (`taux1` ou `taux2` respectivement), et enfin appellent `synchronize_zones` pour mettre à jour les montants affichés dans les zones d'entrée. Les taux de conversion utilisent le franc comme devise de référence.

```
# pack [barre_de_menus] [Side Side_Top; Fill Fill_X];
  pack [ligne1; ligne2] [Side Side_Top; Fill Fill_X];
  mainLoop ();;
```

11.4 Le jeu du taquin

Pour conclure ce chapitre, nous écrivons en CamlTk un jeu de taquin. Le taquin est un puzzle inventé en 1879 par Sam Lloyd et constitué de pièces rectangulaires. L'une des pièces manque, ce qui permet de déplacer les autres pièces en les faisant glisser dans l'espace ainsi ménagé. Le but du jeu est bien sûr de reconstituer l'image en faisant ainsi glisser les pièces.



Ce petit jeu est l'occasion d'introduire un nouvel outil de placement des composants : la toile (*canvas*). Jusqu'ici, tous nos placements de composants s'effectuaient par la fonction `pack`, qui empile les composants les uns sur les autres. L'utilisation d'une toile permet de placer les composants en donnant leurs coordonnées (x, y) à l'intérieur de la toile. C'est particulièrement utile pour manipuler des composants de nature géométrique, tels que polygones, ovales, ou images numérisées (*bitmaps*). Les coordonnées des composants dans la toile peuvent être changées à tout instant, ce qui permet de les déplacer à l'écran.

```
# let découpe_image img nx ny =
    let l = imagephoto__width img
    and h = imagephoto__height img in
    let tx = l / nx and ty = h / ny in
    let pièces = ref [] in
    for x = 0 to nx - 1 do
      for y = 0 to ny - 1 do
        let pièce = imagephoto__create
                      [Width (Pixels tx); Height (Pixels ty)] in
        imagephoto__copy pièce img
          [ImgFrom(x * tx, y * ty, (x+1)*tx, (y+1)*ty)];
        pièces := pièce :: !pièces
      done
    done;
    (tx, ty, tl !pièces);;
```

Nous commençons par une fonction qui charge une image au format GIF depuis un fichier et la découpe en pièces rectangulaires. Les paramètres `nx` et `ny` donnent le nombre de pièces horizontalement et verticalement. À l'aide des fonctions de manipulations d'image fournies par le module `imagephoto` de CamlTk, la fonction `découpe_image` charge l'image depuis le fichier et la découpe en `nx * ny` petites images rectangulaires. La première de ces images est alors abandonnée pour laisser de la place au « trou » du jeu de taquin ; les autres sont renvoyées en résultat, et vont constituer les pièces du taquin.

La fonction `remplir_taquin` ci-dessous se charge de positionner les pièces du taquin dans une toile rectangulaire `c` passée en argument. Elle associe ensuite à l'événement « clic souris » dans les pièces une fonction Caml qui permet de déplacer les pièces.

Nous créons un composant `trou` de type rectangle pour représenter l'emplacement restant libre sur le taquin. Ses coordonnées sont conservées dans les références `trou_x` et `trou_y`. La matrice (tableau bidimensionnel) `taquin` associe à chaque coordonnée (x, y)

le composant représentant la pièce qui se trouve à cet endroit. Nous la remplissons avec la liste de pièces passée en argument. Comme il manque exactement une pièce pour remplir tout le jeu, la dernière case de la matrice reste égale à `trou`. Chaque image de la liste `pièces` est transformée en élément de toile et placée à sa position initiale par la fonction `canvas__create_image`. Enfin, l'option `Tags [Tag "pièce"]` associe à chaque image le nom symbolique `pièce` ; nous l'utiliserons plus tard pour associer une action à l'événement « clic souris » dans tous les pièces d'un coup.

```
# let remplir_taquin c nx ny tx ty pièces =
    let trou_x = ref (nx - 1)
    and trou_y = ref (ny - 1) in
    let trou =
      canvas__create_rectangle c
        (Pixels (!trou_x * tx)) (Pixels (!trou_y * ty))
        (Pixels tx) (Pixels ty) [] in
    let taquin = make_matrix nx ny trou in
    let p = ref pièces in
    for x = 0 to nx - 1 do
      for y = 0 to ny - 1 do
        match !p with
        | [] -> ()
        | pièce :: reste ->
            taquin.(x).(y) <-
              canvas__create_image c
                  (Pixels (x * tx)) (Pixels (y * ty))
                  [ImagePhoto pièce; Anchor NW; Tags [Tag "pièce"]];
              p := reste
      done
    done;
    let déplacer x y =
      let pièce = taquin.(x).(y) in
      canvas__coords_set c pièce
        [Pixels (!trou_x * tx); Pixels(!trou_y * ty)];
      canvas__coords_set c trou
        [Pixels (x * tx); Pixels(y * ty); Pixels tx; Pixels ty];
      taquin.(!trou_x).(!trou_y) <- pièce;
      taquin.(x).(y) <- trou;
      trou_x := x; trou_y := y in
    let jouer ei =
      let x = ei.ev_MouseX / tx and y = ei.ev_MouseY / ty in
      if x = !trou_x && (y = !trou_y - 1 || y = !trou_y + 1)
      || y = !trou_y && (x = !trou_x - 1 || x = !trou_x + 1)
      then déplacer x y in
    canvas__bind c (Tag "pièce") [[], ButtonPress]
                  (BindSet ([Ev_MouseX; Ev_MouseY], jouer));;
```

La fonction `déplacer` ci-dessus prend la pièce en position (x, y) et la fait glisser à la place du trou. Elle suppose que la pièce (x, y) est adjacente au trou. Elle se contente d'échanger les coordonnées de la pièce et celles du trou, tout en effectuant la même permutation dans la matrice `taquin`.

L'appel à `canvas__bind` assure que la fonction `jouer` est appelée à chaque fois

que l'utilisateur clique sur un des éléments de la toile qui porte le nom symbolique `pièce`. c'est-à-dire sur l'une des images composant les pièces du taquin. La fonction `jouer` détermine les coordonnées du clic souris à partir de l'enregistrement `ei` fourni par CamlTk, vérifie que le clic porte bien sur une pièce adjacente au trou, et finalement déplace cette pièce.

```
# let rec permutation = function
    | [] -> []
    | l  -> let n = random__int (list_length l) in
            let (élément, reste) = partage l n in
            élément :: permutation reste
  and partage l n =
    match l with
    | [] -> failwith "partage"
    | tête :: reste ->
        if n = 0 then (tête, reste) else
          let (élément, reste') = partage reste (n - 1) in
          (élément, tête :: reste');;
```

Pour rendre le jeu intéressant, il faut mélanger initialement les pièces. La fonction `permutation` ci-dessus effectue une permutation aléatoire d'une liste. Elle choisit au hasard un élément de la liste, puis permute récursivement les autres éléments de la liste, et remet l'élément choisi en tête du résultat.

```
# let taquin nom_fichier nx ny =
    let fenêtre_principale = openTk () in
    let img = imagephoto__create [File nom_fichier] in
    let c = canvas__create fenêtre_principale
            [Width(Pixels(imagephoto__width img));
             Height(Pixels(imagephoto__height img))] in
    let (tx, ty, pièces) = découpe_image img nx ny in
    remplir_taquin c nx ny tx ty (permutation pièces);
    pack [c] [];
    mainLoop ();;
```

11.5 Pour aller plus loin

CamlTk est une bibliothèque d'une grande richesse, et la présentation que nous en avons faite dans ce chapitre est forcément incomplète. Nous espérons cependant qu'elle aura convaincu le lecteur de la facilité avec laquelle on peut doter une application Caml d'une interface homme-machine de bonne facture.

Pour une présentation plus complète de la boîte à outils Tk, on se reportera à l'ouvrage de John Ousterhout, *Tcl and the Tk toolkit*, Addison-Wesley, ou à celui de Matt Welch, *Practical programming in Tcl and Tk*, Prentice-Hall.

II

Exemples complets

Avertissement

Enfin de vrais programmes !
— Pourquoi vrais ? Les programmes précédents étaient-ils faux ?

Dans la première partie de ce livre, nous avons abordé tous les mécanismes essentiels du langage Caml. Dans cette deuxième partie, nous développons des exemples de programmes complets, dans le but de montrer comment appliquer toute la puissance du langage à la résolution de problèmes de programmation. Nos programmes sont de « vrais » programmes : ils sont issus de problèmes ayant une réelle importance pratique et se présentent sous la forme d'applications indépendantes, utilisables en dehors du système interactif.

Chaque chapitre commence par une description intuitive de ce que le programme d'exemple doit faire. Nous introduisons ensuite les notions nécessaires pour rendre cette spécification précise, ainsi que les principaux algorithmes résolvant le problème. Nous implémentons ensuite la solution en Caml, sous la forme d'un programme indépendant découpé en modules. Les chapitres se terminent par un paragraphe « Pour aller plus loin », contenant en particulier des références bibliographiques à l'intention du lecteur qui désire en savoir plus.

Les exemples proviennent de divers domaines de l'informatique et supposent que le lecteur est déjà vaguement familier avec les domaines concernés. Notre but n'est pas de fournir une introduction complète, partant de zéro, à ces domaines (plusieurs livres n'y suffiraient pas), mais de montrer le langage Caml à l'œuvre. Les notions supposées connues font cependant partie de la culture informatique de base. Par exemple, le chapitre consacré à la simulation d'un microprocesseur suppose quelques notions élémentaires d'architecture des machines et une expérience (même minime) de la programmation en assembleur. De même, le chapitre consacré à la compilation d'un langage impératif simplifié suppose quelques connaissances en Pascal. À chaque fois, nous fournirons des explications sur le problème précis qui nous intéresse et sur les techniques exactes que nous appliquerons ; mais nous supposerons connue la problématique d'ensemble.

Nous utiliserons la présentation suivante pour les programmes Caml : les interfaces et les implémentations des modules sont entremêlées avec nos commentaires ; chaque morceau de code Caml est surtitré par le nom du fichier auquel il appartient. Par exemple, voici un morceau du fichier `toto.ml` :

Fichier `toto.ml`

```
let x = 1 + 2;;
```

On reconstitue le programme tout entier en recollant tous les morceaux de fichiers dans leur ordre d'apparition. À la fin de chaque chapitre, on explique comment compiler et exécuter le programme complet.

Comme pour la première partie, tous les exemples présentés ont été automatiquement extraits du texte, puis compilés et exécutés. Enfin, ces exemples sont disponibles sur le Web à l'adresse suivante : `http://caml.inria.fr/Examples/`.

12
Démonstration de propositions

Mais ou et donc or ni car non si alors ...

DÉCANISER en partie le raisonnement mathématique et transformer ainsi les ordinateurs en outils d'aide à la démonstration de théorèmes est l'un des plus vieux projets de l'informatique. Dans ce chapitre, nous programmons un démonstrateur pour une classe restreinte de théorèmes, les formules propositionnelles du premier ordre. Ce chapitre constitue également une initiation à la logique mathématique élémentaire. De surcroît, nous compléterons nos connaissances sur l'analyse syntaxique et lexicale, en introduisant un générateur d'analyseurs lexicaux et les concepts de mot-clé réservé et de priorité des opérateurs. Pour expliquer le générateur d'analyseurs lexicaux, nous serons aussi obligés d'introduire les tables de hachage, une structure de données très utile.

12.1 La logique mathématique

La logique mathématique traite de la véracité des phrases mathématiques et de la validité des raisonnements. Elle permet de répondre à des questions comme : sachant que la phrase P est vraie et que la phrase Q est fausse, est-ce que la phrase obtenue en disant « P et Q » est une phrase vraie ? Un exemple plus complexe : le raisonnement suivant est-il correct ?

> Puisque P est vraie et que Q est fausse,
> P n'est donc pas une condition nécessaire pour que Q soit vraie.

(Réponse à la section 12.7.) La logique mathématique permet de répondre à ces questions en définissant précisément les opérations autorisées sur les phrases mathématiques et la signification de ces opérations. Une opération licite est par exemple de relier deux phrases mathématiques par « et », comme dans « 2 est pair *et* 2 est un nombre premier ». La logique mathématique fournit donc un sens précis à tous les petits mots qu'on emploie dans les raisonnements, comme « donc », « or », « car », « et », « ou », etc.

La logique définit aussi un calcul sur les phrases mathématiques, indépendant de leur signification réelle, qui ne s'attache qu'à la vérité des phrases. On déduit ainsi la vérité d'une phrase complexe par un simple calcul, à partir de la vérité de ses composantes.

Par exemple, sachant que P est vraie et que Q est fausse, on saura calculer si le contraire de la phrase « P ou Q » est vrai ou faux.

Les propositions

La première étape est donc de définir ce qu'on entend par phrase mathématique. En effet, tout énoncé n'est pas forcément une phrase mathématique. Le critère minimal est la non-contradiction : une phrase mathématique peut être vraie ou fausse, mais on exige qu'elle ne soit pas à la fois vraie *et* fausse. Un exemple paradigmatique de phrase contradictoire est le paradoxe du menteur : c'est la simple phrase « Je mens ». En effet, cette phrase est à la fois vraie et fausse. On le démontre facilement en la supposant d'abord vraie et en montrant qu'elle est alors forcément fausse, puis en la supposant fausse et en montrant qu'alors elle est vraie.

1. Supposons la phrase vraie. C'est donc qu'il est vrai que la personne qui parle ment, qu'elle ne dit pas la vérité. Donc cette personne énonce des phrases fausses et donc la phrase qu'elle vient d'énoncer, « Je mens », est fausse.
2. Supposons la phrase fausse. La phrase « Je mens » est donc inexacte. C'est donc que le locuteur ne ment pas et dit la vérité. C'est donc que la phrase qu'il énonce est vraie.

Dans les deux cas, la phrase est à la fois vraie et fausse : ce n'est donc pas une phrase mathématique.

Une phrase acceptable est appelée *proposition*. Une proposition peut donc être soit vraie soit fausse. Les valeurs « vrai » et « faux » sont appelées valeurs de vérité ; nous les noterons en abrégé v et f.

Pour ceux qui penseraient — à tort — que le paradoxe du menteur vient sans doute de confusions dues à l'imprécision du langage courant, on peut prendre d'autres exemples, formulés dans un langage tout à fait mathématique cette fois-ci. Par exemple : « l'ensemble de tous les ensembles qui ne sont pas éléments d'eux-mêmes est élément de lui-même ». Par le même raisonnement que pour le paradoxe du menteur, on voit que cet énoncé est à la fois vrai et faux. Notez le lien avec les définitions récursives qui ne sont pas bien fondées : nous avons déjà souligné qu'une phrase qui présente des autoréférences est susceptible de mener au non-sens, si l'on n'y prend pas garde (section 2.1).

Les connecteurs propositionnels

Les opérations que nous allons définir sur les propositions sont appelées connecteurs propositionnels, puisqu'elles relient des propositions pour fabriquer d'autres propositions. Nous commençons par le « contraire » d'une proposition, qu'on appelle aussi sa négation. C'est le connecteur « non » : si P est une proposition, alors *non* P est une proposition, qui est fausse si P est vraie et qui est vraie si P est fausse. Techniquement, la proposition *non* P est souvent notée $\neg P$ ou encore $\overline{P}$. Pour ne pas multiplier les notations, nous n'utiliserons pas les noms techniques des connecteurs, mais leur nom vulgaire. Ainsi, nous continuerons à noter *non* le connecteur $\neg$. On définit formellement le connecteur *non* en envisageant toutes les valeurs de vérité possibles de son argument

et en donnant pour chacune d'elles la valeur de vérité correspondante de *non* P. On écrit donc cette définition sous la forme d'un tableau de cas, qu'on nomme « table de vérité » :

P	*non* P
v	f
f	v

Cette table comprend sur chaque colonne une proposition et ses différentes valeurs de vérité possibles. La première ligne dresse ainsi la liste de toutes les propositions décrites par la table. Les autres lignes donnent les valeurs de vérité de toutes les propositions de façon cohérente, c'est-à-dire selon les valeurs prises par les propositions de base. La table précédente comprend donc deux lignes, puisqu'il y a deux cas possibles pour P. La deuxième ligne indique donc que lorsque P vaut v, *non* P vaut f, et la troisième ligne que *non* P vaut v quand P vaut f.

Nous définissons maintenant des opérations binaires, le « et » (la conjonction) et le « ou » (la disjonction). Si P est une proposition et Q est une proposition, alors P *et* Q est une proposition. Par définition, P *et* Q n'est vraie que si P et Q sont simultanément vraies. La proposition P *et* Q est notée $P \wedge Q$ en mathématiques. L'opération *et* est aussi définie par une table de vérité, mais le tableau comporte plus de lignes que pour l'opération *non*, car il faut envisager tous les cas possibles pour P et pour Q, c'est-à-dire quatre cas.

P	Q	P *et* Q
v	v	v
v	f	f
f	v	f
f	f	f

Remarquez que P *et* Q est fausse dès que l'une des propositions P ou Q est fausse.

Le « ou » est symétrique du « et », en remplaçant vrai par faux : par définition, P *ou* Q n'est fausse que si P et Q sont simultanément fausses. La proposition P *ou* Q est notée $P \vee Q$ en mathématiques.

P	Q	P *ou* Q
v	v	v
v	f	v
f	v	v
f	f	f

Remarquez que P *ou* Q est vraie dès que l'une des propositions P ou Q est vraie.

Ce sont les seules définitions nécessaires en logique élémentaire. Toutes les autres constructions du raisonnement s'expriment en fonction de celles-ci. Cette économie de moyens est l'une des beautés de la logique. Toute la théorie est construite sur les notions élémentaires et intuitives de valeur de vérité, de « et », de « ou » et de « non ».

Remarques sur les connecteurs « et » et « ou »

Il faut noter que le « ou » de la logique est inclusif, c'est-à-dire que P *ou* Q est encore vraie si P et Q sont toutes les deux vraies. Nous venons de le voir, P *ou* Q est vraie

dès que l'une des propositions est vraie ; si les deux propositions P et Q sont vraies, P *ou* Q est *a fortiori* vraie, on serait tenté de dire « encore plus vraie », puisqu'il y a deux raisons pour qu'elle soit vraie. La difficulté vient de ce que l'emploi de « ou » dans la langue commune n'est pas toujours celui-là. En réalité la sémantique du « ou » dans la langue parlée est assez floue. Considérez les phrases suivantes :

Fromage ou dessert.

Défense de fumer ou de cracher.

Mange ta soupe ou tu auras une fessée.

Dans « fromage ou dessert » le ou est exclusif : on aura du fromage, ou bien du dessert, mais pas les deux. En revanche, le « ou » de « fumer ou cracher » a le sens des mathématiques : il est inclusif. En effet, il est interdit de fumer, il est interdit aussi de cracher, mais il est « encore plus » interdit de cracher tout en fumant. Finalement, le « ou » de « mange ta soupe ou tu auras une fessée » a le sens d'une déduction ; on pourrait le remplacer par sinon : « mange ta soupe sinon tu auras une fessée ». C'est le sens aussi d'un « si alors » : « si tu ne manges pas ta soupe alors tu auras une fessée ». Cette signification particulière du « ou » n'est pas un hasard, c'est au contraire l'exacte définition mathématique de l'implication. Le raisonnement *si* P *alors* Q est synonyme de (*non* P) *ou* Q. Par exemple « si nous ne nous hâtons pas nous serons en retard » est synonyme de « hâtons-nous ou nous serons en retard ».

L'implication

Le raisonnement « si alors » est appelé implication en logique. L'implication est traditionnellement notée $\Rightarrow$: si P et Q sont des propositions, alors par définition $P \Rightarrow Q$ est une proposition, qui a la même valeur de vérité que (*non* P) *ou* Q. La proposition $P \Rightarrow Q$ se lit « P implique Q ».

Nous avons maintenant défini précisément toutes les opérations de base sur les propositions. Il nous reste à montrer les méthodes de calcul sur ces opérations.

12.2 Calculs de tables de vérité

Nous commençons par un calcul « à la main », qui nous permettra de comprendre comment va marcher le programme de démonstration automatique.

Une première démonstration

Nous allons établir la table de vérité de l'implication. Plus précisément, nous démontrons que la proposition $P \Rightarrow Q$ a la table de vérité suivante :

P	Q	$P \Rightarrow Q$
v	v	v
v	f	f
f	v	v
f	f	v

Nous avons défini $P \Rightarrow Q$ comme $(non\ P)\ ou\ Q$, ce qui signifie que ce sont les mêmes propositions, ou encore qu'elles ont toujours la même valeur de vérité, quelles que soient les valeurs de vérité des propositions P et Q. Le calcul de la table de vérité de l'implication est donc très simple : on procède par étapes dans une table de vérité où l'on a énuméré toutes les possibilités pour P et Q : on calcule d'abord la proposition $non\ P$ dans tous les cas de figures, puis le « ou » de $non\ P$ et de Q. On obtient finalement :

P	Q	$non\ P$	$(non\ P)\ ou\ Q$
v	v	f	v
v	f	f	f
f	v	v	v
f	f	v	v

Constatez que l'implication est donc vraie si l'hypothèse est fausse (deux dernières lignes du tableau). Ceci correspond à l'intuition : lorsqu'on a un théorème vrai $P \Rightarrow Q$, mais qu'on n'a pas l'hypothèse P, alors on ne peut évidemment rien en déduire sur Q, puisque dans ce cas Q peut aussi bien être vraie que fausse (toujours les deux dernières lignes du tableau). D'autre part, il est impossible d'attribuer une autre valeur de vérité à l'implication lorsque l'hypothèse n'est pas vraie. En effet, si l'on mettait f à la place de v dans les deux dernières lignes de la colonne de $(non\ P)\ ou\ Q$, cela signifierait qu'un théorème $P \Rightarrow Q$ devient faux dès que son hypothèse est fausse, ce qui serait absurde. On résume parfois cette situation en disant « le faux implique n'importe quoi » ; nous préférons la formulation « on ne peut rien déduire d'un théorème dont l'hypothèse n'est pas vérifiée », ou encore « un théorème reste vrai même quand il ne s'applique pas ».

L'équivalence

Vous connaissez sans doute déjà un autre connecteur propositionnel dont nous n'avons pas parlé : le « si et seulement si », qu'on appelle l'équivalence et qu'on note d'habitude $\Leftrightarrow$. Intuitivement, deux propositions sont équivalentes quand elles sont toujours vraies ou fausses en même temps. Par exemple, si $P \Leftrightarrow Q$ est vraie, on déduit immédiatement la valeur de vérité de Q si l'on connaît celle de P : quand P est vraie on en déduit que Q est vraie et quand P est fausse on en déduit que Q est fausse.

Nous n'avons pas défini ce connecteur car il s'exprime en fonction de ceux que nous connaissons déjà. Vous savez peut-être que $P \Leftrightarrow Q$ est une double implication, comme le suggère la double flèche. En effet $P \Leftrightarrow Q$ signifie que P implique Q (la proposition *directe*) et que de plus Q implique P (la proposition *réciproque*). En fait, on définit l'équivalence par cette propriété : la proposition $P \Leftrightarrow Q$ vaut, par définition, ce que vaut la proposition $(P \Rightarrow Q)\ et\ (Q \Rightarrow P)$. Comme ci-dessus, nous établissons la table de vérité de l'équivalence, en procédant par étapes au calcul de la table de vérité de la proposition complexe $(P \Rightarrow Q)\ et\ (Q \Rightarrow P)$. Nous obtenons :

P	Q	$P \Rightarrow Q$	$Q \Rightarrow P$	$(P \Rightarrow Q)\ et\ (Q \Rightarrow P)$
v	v	v	v	v
v	f	f	v	f
f	v	v	f	f
f	f	v	v	v

Cette table de vérité correspond à l'intuition : l'équivalence de deux propositions n'est vraie que dans le cas où les deux propositions ont la même valeur de vérité. D'un point de vue logique, des propositions équivalentes sont indiscernables : elles sont donc égales au sens logique. En effet, la logique ne distingue les propositions que par leur valeur de vérité, pas par leur texte. Deux propositions équivalentes ne sont donc pas logiquement différentes. C'est le même phénomène qu'en arithmétique, où l'on ne peut distinguer $1 + 1$ de 2, bien que ces deux expressions ne soient pas syntaxiquement les mêmes.

On sent bien que le calcul des tables de vérité est automatisable et qu'un programme remplirait facilement les colonnes des tableaux à notre place, en calculant ainsi la valeur de vérité d'une proposition complexe en fonction des propositions qui la composent. Le programme engendrerait plus facilement que nous toutes les combinaisons possibles pour les différentes propositions élémentaires qui interviennent. En effet cette combinatoire augmente très vite : pour 2 propositions P et Q nous avons 4 lignes dans le tableau, mais pour 3 il en faudrait 8, pour 4 propositions 16 lignes, et ainsi de suite (pour n propositions 2^n lignes). Notre démonstrateur de théorèmes fonctionne exactement selon ce principe.

12.3 Le principe des démonstrations

Notre démonstrateur est restreint aux théorèmes de logique élémentaire. Or, un théorème n'est rien d'autre qu'une proposition qui est toujours vraie. Ceci conduit d'ailleurs à une petite difficulté : en mathématiques, on n'écrit jamais qu'une proposition est vraie. On se contente de l'écrire simplement, ce qui sous-entend qu'elle est vraie. On écrira par exemple :

> Si un triangle a trois angles de 60 degrés alors ses trois côtés ont même longueur.

et non pas :

> L'implication « si un triangle ... alors ... » est vraie.

En revanche, on précise explicitement qu'une proposition est fausse.

Le but du démonstrateur est donc de montrer qu'une proposition est toujours vraie. Pour cela, il démontre que pour toutes les valeurs possibles des propositions élémentaires, la proposition à démontrer a toujours la valeur de vérité « vrai ». Conceptuellement, cela correspond à établir la table de vérité de la proposition et à vérifier que la colonne de droite, celle de la proposition à démontrer, ne contient que des valeurs *v*.

En logique, une proposition toujours vraie est aussi appelée une *tautologie*. Nous dirons donc que notre programme est un démonstrateur de tautologies. Nous lui ajoutons un raffinement supplémentaire : au cas où la proposition qui lui est soumise n'est pas une tautologie, le programme renvoie une *réfutation*, c'est-à-dire un choix de valeurs de vérité pour les propositions élémentaires qui rend fausse la proposition soumise.

L'architecture du programme suit exactement la méthode des tables de vérité : pour calculer les valeurs de vérité des lignes de la table nous allons écrire un sous-programme qui calcule la valeur de vérité d'une proposition en fonction de ses composantes (les

« et », « ou », « non » qui interviennent) ; ensuite, pour produire l'ensemble des lignes de la table, nous écrirons une autre fonction qui envisagera l'ensemble des valeurs possibles des propositions élémentaires. Pour simplifier l'utilisation du démonstrateur, nous ferons finalement un analyseur syntaxique, qui nous permettra d'entrer facilement les propositions. Cela devient presque de la routine maintenant : nous définirons le type des propositions et l'analyseur syntaxique analysera une chaîne de caractères, qu'il transformera en une valeur du type des propositions.

12.4 Représentation et vérification des propositions

Nous commençons par le module `prop`, qui définit le type des propositions et implémente les fonctions d'évaluation d'une proposition et de génération de la combinatoire des possibilités décrites ci-dessus.

Le type des propositions

Le type des propositions comporte des constructeurs pour les connecteurs de base `Non`, `Et`, `Ou`, et des constructeurs pour les connecteurs définis à partir des connecteurs de base, `Implique` et `Équivalent`. Pour représenter les propositions élémentaires comme P ou Q, qu'on appelle aussi variables propositionnelles, on fournit le constructeur `Variable` qui prend en argument une chaîne de caractères (le nom de la proposition). Ces variables représentent les propositions dont on ne connaît pas la valeur et dont les différentes valeurs de vérité forment les lignes des tables de vérité. Une variable est donc une proposition indéfinie, représentée par un simple nom. Une variable qui intervient dans une proposition R est dite *variable libre* de R. Par exemple, P est une variable libre de la proposition P *ou* Q. Plus précisément, la proposition P *ou* Q comporte deux variables libres, P et Q.

Pour des raisons de commodité, on a ajouté deux constructeurs constants particuliers `Vrai` et `Faux`, qui représentent deux propositions particulières, la proposition toujours vraie et la proposition toujours fausse. Ces propositions correspondent aux valeurs de vérité, considérées comme des propositions.

Fichier `prop.mli`

```
type proposition =
   | Vrai
   | Faux
   | Non of proposition
   | Et of proposition * proposition
   | Ou of proposition * proposition
   | Implique of proposition * proposition
   | Équivalent of proposition * proposition
   | Variable of string;;

exception Réfutation of (string * bool) list;;
value vérifie_tautologie: proposition -> string list -> unit
  and variables_libres: proposition -> string list;;
```

Le module `prop` exporte deux fonctions : `vérifie_tautologie`, qui vérifie qu'une proposition est une tautologie ou sinon déclenche l'exception `Réfutation`, et `variables_libres`, qui calcule la liste des variables libres d'une proposition.

L'évaluateur de propositions

Une proposition comporte donc parfois des variables libres. Mais pour calculer la valeur de vérité d'une proposition, il faut absolument connaître la valeur de ses variables libres. Comme d'habitude, notre machine ne peut calculer qu'en connaissant la valeur de toutes les entités qui interviennent dans l'expression à calculer (cf. l'exemple « $x - x$ » de la section 1.3). Nous utiliserons donc des liaisons qui associent une valeur booléenne à toute variable libre de la proposition étudiée. Étant donné un ensemble de liaisons de variables, l'évaluation d'une proposition s'opère en traduisant simplement les connecteurs de la logique à l'aide des opérateurs logiques de Caml : `not`, `&&`, `||`, `=`.

Fichier `prop.ml`

```
let rec évalue_dans liaisons = function
  | Vrai -> true
  | Faux -> false
  | Non p -> not (évalue_dans liaisons p)
  | Et (p, q) -> (évalue_dans liaisons p) &&  (évalue_dans liaisons q)
  | Ou (p, q) -> (évalue_dans liaisons p) || (évalue_dans liaisons q)
  | Implique (p, q) ->
      (not (évalue_dans liaisons p)) || (évalue_dans liaisons q)
  | Équivalent (p, q) ->
      évalue_dans liaisons p = évalue_dans liaisons q
  | Variable v -> assoc v liaisons;;
```

La fonction `évalue_dans` oblige donc à faire la relation entre les connecteurs de la logique et les opérateurs logiques de Caml. Il faudrait démontrer, en analysant tous les cas possibles des propositions arguments des connecteurs, que le « non » de la logique correspond bien au `not` de Caml, de même que « et » correspond à `&&` et « ou » à `||`. Nous supposerons cette démonstration faite, dans la mesure où cette traduction est très naturelle et intuitive. L'implication est évaluée en utilisant sa définition : $P \Rightarrow Q$ se calcule comme (*non* P) *ou* Q. L'équivalence est évaluée en vérifiant simplement que ses deux arguments ont la même valeur de vérité.

Le vérificateur de propositions

Le principe du vérificateur est d'évaluer la proposition pour toutes les valeurs possibles de ses variables libres : si l'une des évaluations renvoie faux, alors la proposition n'est pas une tautologie. On signale ce fait en déclenchant l'exception `Réfutation` avec pour argument une liste de paires (chaîne, booléen) décrivant l'ensemble des valeurs attribuées aux variables qui ont permis de calculer la valeur de vérité f pour la proposition. Cette liste constitue bien une réfutation de la proposition, puisqu'elle montre par un exemple que la proposition n'est pas universellement vraie.

Fichier `prop.ml`

```
let rec vérifie_lignes proposition liaisons variables =
  match variables with
  | [] ->
     if not évalue_dans liaisons proposition
     then raise (Réfutation liaisons)
  | var :: autres ->
     vérifie_lignes proposition ((var, true) :: liaisons) autres;
     vérifie_lignes proposition ((var, false):: liaisons) autres;;

let vérifie_tautologie proposition variables =
  vérifie_lignes proposition [] variables;;
```

La fonction `vérifie_lignes` vérifie toutes les lignes de la table de vérité, sans la construire effectivement. Elle prend en argument une proposition, un ensemble de liaisons et la liste des variables libres de la proposition. Elle lie alors les variables libres à des valeurs `true` ou `false`, puis évalue la proposition. En effet, la règle `[] ->` procède à l'évaluation de la proposition, lorsqu'il n'y a plus de variables à lier. La seconde règle correspond au cas où il y a des variables à lier ; elle exécute une séquence de deux appels récursifs à `vérifie_lignes`, en liant la première variable rencontrée d'abord à `true`, puis à `false`. Ce programme assure donc que toutes les combinaisons possibles seront envisagées et si la vérification ne déclenche jamais l'exception `Réfutation` on aura effectivement prouvé que la proposition s'évalue toujours en `true` dans toutes les liaisons possibles de ses variables. La fonction `vérifie_tautologie` se contente d'appeler `vérifie_lignes` avec un ensemble de liaisons initialement vide.

Dans un style apparemment plus « fonctionnel », on écrirait :

```
let rec vérifie_lignes proposition liaisons = function
  | [] ->
     évalue_dans liaisons proposition || raise (Réfutation liaisons)
  | var :: autres ->
     vérifie_lignes proposition ((var, true) :: liaisons) autres &&
     vérifie_lignes proposition ((var, false):: liaisons) autres;;
```

Cette version n'est pas plus claire que la précédente : elle est trompeuse car bien qu'elle semble calculer un booléen, son résultat n'est pas intéressant. En effet, elle retourne toujours le booléen `true` si la proposition est une tautologie, ou lève une exception si la proposition est réfutable. C'est donc bien une procédure, puisqu'elle fonctionne par effets : l'effet attendu est soit « évaluation réussie », soit un déclenchement d'exception. Il ne sert à rien de la déguiser en fonction ... Si l'on renonce à renvoyer une réfutation de la proposition analysée, il est possible d'écrire une vraie fonction qui calcule vraiment un booléen. Malheureusement on perd la liaison des variables qui a prouvé que la proposition n'est pas une tautologie et il faut alors écrire une autre fonction, complètement analogue, pour renvoyer une réfutation. Cet exemple nous montre un autre intérêt des exceptions : dans certains cas une fonction peut calculer en fait deux résultats de type différent, l'un véhiculé par le mécanisme normal des appels de fonctions, l'autre transporté par une exception (`vérifie_lignes` calcule un booléen dans le cas d'une tautologie et une liste d'association (nom de variable, valeur booléenne) dans le cas contraire).

Une remarque de complexité : comme nous l'avons déjà vu, le nombre de lignes d'une table de vérité est 2^n, où n est le nombre de variables qui interviennent dans la table. Notre fonction `vérifie_tautologie` suit exactement la méthode des tables de vérité ; elle a donc une complexité exponentielle. Ce n'est pas très grave pour nous, car nous nous limiterons à trois variables au plus. C'est un problème actuellement ouvert que de savoir s'il existe des algorithmes d'une meilleure complexité que le nôtre pour calculer la table de vérité d'une formule. Dans le pire des cas on ne sait évidemment pas faire mieux, mais dans certains cas particuliers, on parvient à calculer les tables de vérité de formules où interviennent des centaines, voire des milliers de variables, ce qui est absolument hors de portée de notre programme. On utilise pour cela des structures sophistiquées de partage de tables de vérité, comme par exemple les *binary decision diagrams*.

Calcul des variables libres

Pour appeler la fonction `vérifie_tautologie`, nous devons disposer d'une fonction qui détermine l'ensemble des propositions élémentaires d'une proposition, ce que nous appelons aussi ses variables libres. La liste des variables libres s'obtient facilement par un parcours récursif de la proposition, à la recherche de sous-propositions de la forme `Variable v`.

Fichier `prop.ml`

```
let rec variables accu proposition =
  match proposition with
  | Variable v -> if mem v accu then accu else v :: accu
  | Non p -> variables accu p
  | Et (p, q) -> variables (variables accu p) q
  | Ou (p, q) -> variables (variables accu p) q
  | Implique (p, q) -> variables (variables accu p) q
  | Équivalent (p, q) -> variables (variables accu p) q
  | _ -> accu;;

let variables_libres proposition = variables [] proposition;;
```

La seule difficulté est que les variables ne doivent pas être répétées dans la liste résultat. Par exemple, si on cherche les variables de la proposition *P et P*, on va rencontrer deux fois le terme `(Variable "P")`. Pourtant, la chaîne `"P"` ne doit apparaître qu'une fois dans le résultat de `variables_libres`. C'est pourquoi, lorsqu'on rencontre `Variable v`, on teste si `v` n'appartient pas déjà à la liste des variables collectées jusqu'à présent (à l'aide de la fonction prédéfinie `mem`, qui se comporte comme la fonction `membre` de la section 7.3). La fonction `variables` maintient donc un accumulateur des variables déjà collectées : c'est l'argument `accu`, qu'on gère soigneusement lors des appels récursifs. Par exemple, dans le cas d'un opérateur binaire, comme `et`, on appelle récursivement `variables` sur le deuxième argument de l'opérateur, mais avec un accumulateur obtenu en collectant les variables du premier argument :

```
Et (p, q) -> variables (variables accu p) q
```

La fonction principale `variables_libres` se réduit à l'appel de `variables` avec un accumulateur initialement vide.

12.5 Syntaxe concrète des propositions

Nous définissons maintenant la syntaxe concrète des propositions, et les fonctions qui transforment la syntaxe concrète en syntaxe abstraite. Reprenant l'approche du chapitre 9, nous allons procéder en deux temps : analyse lexicale, pour obtenir une suite de lexèmes à partir d'une suite de caractères, puis analyse syntaxique, pour construire un arbre de syntaxe abstraite à partir d'une suite de lexèmes.

L'analyseur lexical

L'analyseur lexical dont nous avons besoin est très proche de celui du chapitre 9 : il doit savoir supprimer les blancs, reconnaître les identificateurs (noms de variables) et distinguer les symboles spéciaux comme les parenthèses. Cependant, nous ne pouvons pas réutiliser tel quel l'analyseur du chapitre 9. D'une part, nous avons besoin de reconnaître des lexèmes formés d'une suite de symboles, comme par exemple `=>` ou `<=>`. D'autre part, nous voulons introduire la notion de mot-clé réservé.

Rappelons qu'un *mot-clé* est une suite de caractères qui a la forme d'un identificateur, mais qui joue un rôle spécial dans le langage, par exemple comme opérateur infixe ou pour introduire des constructions du langage. Ainsi, `if`, `then`, `else` sont des mots-clés du langage Caml. De même, `vrai`, `faux`, `et`, `ou`, `non` sont des mots-clés de la syntaxe concrète des propositions.

Un mot-clé est dit *réservé* s'il ne peut pas être employé comme identificateur. Par exemple, en Caml, il est impossible d'employer `if` comme un nom de variable : une phrase comme `let if = 3` est syntaxiquement incorrecte. Le mot-clé `if` est donc réservé en Caml (ainsi que tous les autres mots-clés). En revanche, le mini-Logo du chapitre 9 n'a pas de mots-clés réservés : on écrit sans problème la procédure

```
pour carré :répète
     répète 4 [av :répète td 90].
```

en utilisant ainsi l'identificateur `répète` à la fois comme mot-clé et comme nom de variable. Dans le cas de mini-Logo ce n'est pas catastrophique, puisque les variables sont explicitement différenciées par le symbole « : » qui les précède. Mais un langage comme PL/1 n'a pas cette convention et pourtant ne réserve aucun mot-clé, autorisant donc des phrases comme :

```
if then = else then else = if else then = if
```

Comme on le voit sur cet exemple, ne pas réserver les mots-clés peut conduire à des programmes vraiment illisibles. Nous allons donc réserver les mots-clés de la syntaxe concrète des propositions. Les changements à apporter à l'analyseur lexical sont minimes : lorsqu'on a reconnu une suite de caractères qui a la forme d'un identificateur, il faut tester si cette suite de caractères est un mot-clé ou non et renvoyer des lexèmes différents selon le cas. Par exemple, ayant lu `truc`, qui n'est pas un mot-clé, on renverra le lexème `(Ident "truc")` ; mais si on lit `ou`, on renverra le lexème `(MC "ou")`. Le constructeur `MC` est l'abréviation de « mot-clé ».

Autant les conventions lexicales de base (qu'est-ce qu'un entier, qu'est-ce qu'un identificateur, ...) sont souvent les mêmes d'un langage à un autre, autant les mots-clés sont hautement spécifiques au langage. Dans le but de rendre notre analyseur lexical réutilisable par la suite, pour d'autres langages que les propositions, nous n'allons

pas mettre « en dur » la liste des mots-clés dans le code de l'analyseur. Au contraire, l'analyseur va prendre en paramètre la liste des mots-clés et renvoyer une fonction d'analyse lexicale (de type `char stream -> lexème stream`) spécialisée pour cette liste de mots-clés. La fonction principale qui fait correspondre une fonction d'analyse à une liste de mots-clés est plus qu'un analyseur lexical, c'est toute une famille d'analyseurs lexicaux en puissance. C'est pourquoi nous l'appelons « générateur d'analyseurs lexicaux », ou « analyseur lexical universel ». Les mots « générateur » et « universel » sont un peu forts car cette fonction impose des conventions lexicales fixes ; elle n'est donc pas « universelle » et ne remplace en aucun cas un véritable générateur d'analyseurs comme `camllex` (cf. le chapitre 7 du *Manuel de référence du langage Caml*). Pourtant cet analyseur lexical « universel » suffira amplement aux besoins de ce livre : c'est le dernier analyseur lexical que nous écrirons, par la suite nous le réutiliserons tel quel pour tous les langages dont nous aurons besoin.

L'analyseur lexical « universel » se présente sous la forme d'un module, `lexuniv`, dont voici l'interface :

Fichier `lexuniv.mli`

```
type lexème =
   | MC of string
   | Ident of string
   | Entier of int;;
value construire_analyseur:
     string list -> (char stream -> lexème stream);;
```

On trouvera une implémentation de ce module à la fin de ce chapitre (section 12.8).

L'analyseur syntaxique

L'analyse syntaxique des propositions ressemble beaucoup à l'analyse syntaxique des expressions dans le mini-Logo. La principale nouveauté est que nous allons prendre en compte les priorités entre opérateurs. Par exemple, vous savez qu'en arithmétique $1+2\times3$ signifie $1+(2\times3)$ et que $1\times2+3$ veut dire $(1\times2)+3$. Les deux écritures sont tolérées et ont la même signification. On dit que l'opérateur $\times$ a une priorité plus élevée que l'opérateur $+$. Cependant, l'analyseur syntaxique du mini-Logo traduit `1*2+3` par

`Produit(Constante 1, Somme(Constante 2, Constante 3))`,

ce qui correspond à $1\times(2+3)$. En effet, il analyse d'abord `1` comme une expression, puis voit l'opérateur `*` et se rappelle donc récursivement pour lire l'expression à la droite du `*`, dont il fait le deuxième argument du constructeur `Produit`. L'analyseur syntaxique du mini-Logo ne tient donc pas compte des priorités relatives des opérateurs.

Nous allons implémenter des conventions de priorité analogues à celles de l'arithmétique pour les opérations de la logique : `et` est plus prioritaire que `ou`, qui est plus prioritaire que `=>`, etc. Dans ce but, l'analyseur syntaxique est stratifié en fonctions qui analysent des propositions de priorité de plus en plus faible. La fonction `proposition0` analyse ainsi les propositions les plus simples, identificateurs, booléens ou expressions entre parenthèses. La fonction `proposition1` analyse les propositions qui commencent par `non`, ou qui sont des propositions simples. Les autres

fonctions `proposition2` à `proposition5` analysent respectivement les propositions qui comportent un `et`, un `ou`, une implication ou une équivalence.

Fichier `asynt.ml`

```
#open "prop";;
#open "lexuniv";;

let rec lire_proposition f = proposition5 f

and proposition0 = function
  | [< 'Ident s >] -> Variable s
  | [< 'MC "vrai" >] -> Vrai
  | [< 'MC "faux" >] -> Faux
  | [< 'MC "("; lire_proposition p; 'MC ")" >] -> p

and proposition1 = function
  | [< 'MC "non"; proposition0 p >] -> Non p
  | [< proposition0 p >] -> p

and proposition2 = function
  | [< proposition1 p; (reste2 p) q >] -> q
and reste2 p = function
  | [< 'MC "et"; proposition1 q; (reste2 (Et (p, q))) r >] -> r
  | [<>] -> p

and proposition3 = function
  | [< proposition2 p; (reste3 p) q >] -> q
and reste3 p = function
  | [< 'MC "ou"; proposition2 q; (reste3 (Ou (p, q))) r >] -> r
  | [<>] -> p

and proposition4 = function
  | [< proposition3 p; (reste4 p) q >] -> q
and reste4 p = function
  | [< 'MC "=>"; proposition3 q; (reste4 (Implique (p, q))) r >] -> r
  | [<>] -> p

and proposition5 = function
  | [< proposition4 p; (reste5 p) q >] -> q
and reste5 p = function
  | [< 'MC "<=>"; proposition4 q; (reste5 (Équivalent(p,q))) r >] -> r
  | [<>] -> p;;
```

Les fonctions `proposition1` à `proposition5` sont toutes construites sur le même moule. Elles cherchent d'abord une proposition de niveau plus simple, puis appellent une fonction `reste`. Cette fonction se charge de détecter l'opérateur, par exemple `et` pour `reste2`, suivi d'une proposition de même niveau et éventuellement d'autres opérateurs du même type (d'autres `et` pour `reste2`). Remarquez que la fonction `reste` prend en argument l'arbre de syntaxe abstraite jusqu'à présent construit et lui ajoute les opérateurs rencontrés ; d'où l'appel `reste2 p` dans la fonction `proposition2` et l'appel récursif `reste2 (Et (p,q))` dans la fonction `reste2`.

La similitude entre les fonctions qui gèrent les priorités des opérateurs suggère d'écrire une fonction générique qui prenne en argument l'opérateur binaire à reconnaître et la fonction de reconnaissance de ses arguments, puis construise automatiquement la fonction `reste` correspondante. C'est possible avec la fonctionnelle `lire_opération` ci-dessous.

Fichier `asynt.ml`

```
let lire_opération lire_opérateur lire_base constructeur =
  let rec lire_reste e1 = function
  | [< lire_opérateur _;
       lire_base e2;
       (lire_reste (constructeur (e1, e2))) e >] -> e
  | [< >] -> e1 in
 function [< lire_base e1; (lire_reste e1) e >] -> e;;
```

L'argument `lire_opérateur` est une fonction qui reconnaît les flux commençant par l'opérateur qui nous intéresse, par exemple

```
function [< 'MC "+" >] -> ()
```

dans le cas de l'opérateur `+`. L'argument `lire_base` est une fonction supposée reconnaître les arguments de l'opérateur. Enfin, l'argument `constructeur` est une fonction qui reçoit les arbres de syntaxe abstraite des arguments et doit construire l'arbre de syntaxe abstraite de l'expression tout entière. En utilisant cette puissante fonctionnelle `lire_opération`, l'analyseur syntaxique se simplifie notablement.

Fichier `asynt.ml`

```
let rec lire_proposition f = proposition5 f

and proposition0 = function
  | [< 'Ident s >] -> Variable s
  | [< 'MC "vrai" >] -> Vrai
  | [< 'MC "faux" >] -> Faux
  | [< 'MC "("; lire_proposition p; 'MC ")" >] -> p

and proposition1 = function
  | [< 'MC "non"; proposition0 p >] -> Non p
  | [< proposition0 p >] -> p

and proposition2 flux =
    lire_opération (function [< 'MC "et" >] -> ())
                   proposition1
                   (function (p,q) -> Et (p,q))
                   flux
and proposition3 flux =
    lire_opération (function [< 'MC "ou" >] -> ())
                   proposition2
                   (function (p,q) -> Ou (p,q))
                   flux
and proposition4 flux =
    lire_opération (function [< 'MC "=>" >] -> ())
                   proposition3
```

```
                  (function (p,q) -> Implique (p,q))
                  flux
and proposition5 flux =
    lire_opération (function [< 'MC "<=>" >] -> ())
                  proposition4
                  (function (p,q) -> Équivalent (p,q))
                  flux;;
```

Pour construire la fonction principale d'analyse syntaxique, nous engendrons un analyseur lexical en appliquant (partiellement) la fonction `construire_analyseur` à la liste des mots-clés, puis nous composons cet analyseur lexical avec la fonction `lire_proposition`.

Fichier `asynt.ml`

```
let analyseur_lexical =
    construire_analyseur
     ["vrai"; "faux"; "("; ")"; "non"; "et"; "ou"; "=>"; "<=>"];;

let analyse_proposition chaîne =
    lire_proposition (analyseur_lexical (stream_of_string chaîne));;
```

Pour finir, nous cachons toutes les fonctions intermédiaires d'analyse syntaxique, en exportant uniquement la fonction principale.

Fichier `asynt.mli`

```
#open "prop";;
value analyse_proposition: string -> proposition;;
```

12.6 Le vérificateur de tautologies

Tout est prêt pour réaliser un démonstrateur de théorèmes se présentant sous la forme d'une boucle d'interaction qui lit des propositions et essaye de les prouver.

Le cœur du démonstrateur est la fonction `examine`. Partant d'une chaîne de caractères, elle la transforme en proposition, calcule ses variables libres et appelle `vérifie_tautologie`. Si rien ne se passe, la proposition est un théorème et la fonction `examine` affiche un message pour le dire. Sinon, `vérifie_tautologie` déclenche l'exception `Réfutation` et `examine` imprime une réfutation prouvant que la proposition fournie n'est pas un théorème.

Fichier `demo.ml`

```
#open "prop";;
#open "asynt";;

let examine chaîne =
    let proposition = analyse_proposition chaîne in
    let variables = variables_libres proposition in
    try
      vérifie_tautologie proposition variables;
      begin match variables with
      | [] ->
```

```
          print_string "Théorème: "
      | [var] ->
          print_string ("Théorème: pour toute proposition "^var^", ")
      | _ ->
          print_string "Théorème: pour toutes propositions ";
          do_list (function var -> print_string (var^", ")) variables
      end;
      print_string chaîne;
      print_newline ()
    with Réfutation liaisons ->
      print_string (chaîne ^ " n'est pas un théorème,\n");
      print_string "car la proposition est fausse quand\n";
      do_list
       (function (var, b) ->
         print_string (var ^ " est ");
         print_string (if b then "vraie" else "fausse");
         print_newline ())
       liaisons;;
```

Nous mettons autour de cette fonction une boucle d'interaction standard, dans le style de celle pour le mini-Logo présentée à la section 10.3.

Fichier `demo.ml`

```
let boucle () =
  try
    while true do
      print_string ">>> "; examine (read_line ())
    done
  with End_of_file -> ();;

if sys__interactive then () else begin boucle (); exit 0 end;;
```

L'identificateur `sys__interactive` est prédéfini par le système, vaut `true` si le programme tourne sous le système interactif et `false` si le programme tourne comme une application indépendante. Nous le testons pour éviter de lancer la boucle d'interaction dans le cas où ce code est exécuté depuis le système interactif. Dans ce cas, il est préférable de laisser l'utilisateur lancer la boucle lui-même et surtout de ne pas sortir prématurément du système interactif par `exit 0` quand la boucle termine.

Compilation du programme

Il ne reste plus qu'à compiler les modules et à les lier entre eux. Voici comment faire, à gauche avec le compilateur indépendant `camlc`, à droite avec le système interactif.

```
$ camlc -c prop.mli            # compile "prop.mli";;
$ camlc -c prop.ml             # compile "prop.ml";;
$ camlc -c lexuniv.mli         # compile "lexuniv.mli";;
$ camlc -c lexuniv.ml          # compile "lexuniv.ml";;
$ camlc -c asynt.mli           # compile "asynt.mli";;
$ camlc -c asynt.ml            # compile "asynt.ml";;
$ camlc -c demo.ml             # compile "demo.ml";;
```

```
$ camlc -o demo prop.zo \       # do_list load_object ["prop.zo";
  lexuniv.zo asynt.zo demo.zo     "lexuniv.zo";"asynt.zo";"demo.zo"];;
```

Le démonstrateur se lance par l'incantation suivante :

```
$ camlrun demo                  #demo__boucle ();;
```

Nous pouvons alors taper des propositions en réponse au signe d'invite « `>>>` ». Pour sortir du programme, il faut taper un caractère « fin de fichier » (`ctrl-D` en Unix) ou une interruption (`ctrl-C` en Unix).

12.7 Exemples de théorèmes

Simples mais tellement vrais

Pour montrer les capacités de notre programme, nous établissons des théorèmes simples, mais de moins en moins intuitifs. Tout d'abord, « le contraire du contraire d'une proposition c'est la proposition elle-même » :

```
>>> non (non P) <=> P
Théorème: pour toute proposition P, non (non P) <=> P
```

Le tiers exclus : une proposition est toujours soit vraie soit fausse mais jamais les deux en même temps.

```
>>> P ou (non P) <=> vrai
Théorème: pour toute proposition P, P ou (non P) <=> vrai
>>> P et (non P) <=> faux
Théorème: pour toute proposition P, P et (non P) <=> faux
```

Dire qu'une proposition est équivalente à « vrai » c'est simplement dire que la proposition est vraie ; inversement, dire qu'elle est équivalente à « faux », c'est dire que sa négation est vraie :

```
>>> (P <=> vrai) <=> P
Théorème: pour toute proposition P, (P <=> vrai) <=> P

>>> (P <=> faux) <=> non P
Théorème: pour toute proposition P, (P <=> faux) <=> non P
```

Les deux théorèmes précédents se réécrivent donc plus simplement :

```
>>> P ou (non P)
Théorème: pour toute proposition P, P ou (non P)
>>> non (P et (non P))
Théorème: pour toute proposition P, non (P et (non P))
```

En logique, il est inutile de répéter deux fois la même chose : « P ou P » c'est P. Et inutile d'insister, « P et P » c'est aussi P.

```
>>> P ou P <=> P
Théorème: pour toute proposition P, P ou P <=> P
>>> P et P <=> P
Théorème: pour toute proposition P, P et P <=> P
```

Il est évident qu'en supposant une hypothèse vraie, on la démontre facilement (« $P \Rightarrow P$ »). De plus, une proposition est toujours équivalente à elle-même.

```
>>> P => P
Théorème: pour toute proposition P, P => P
>>> P <=> P
Théorème: pour toute proposition P, P <=> P
```

On sait bien que $P \Leftrightarrow Q$ est synonyme de $Q \Leftrightarrow P$, mais nous le prouvons :

```
>>> (P <=> Q) <=> (Q <=> P)
Théorème: pour toutes propositions Q, P, (P <=> Q) <=> (Q <=> P)
```

On sait aussi que prouver l'équivalence de deux propositions est équivalent à prouver l'équivalence de leur négation :

```
>>> (non P <=> non Q) <=> (P <=> Q)
Théorème: pour toutes propositions Q, P,
   (non P <=> non Q) <=> (P <=> Q)
```

Mais on connaît souvent moins bien la notion de proposition contraposée d'une implication. La contraposée de $P \Rightarrow Q$ est la proposition $(non\ Q) \Rightarrow (non\ P)$. Elle est intéressante car elle est équivalente à la proposition de départ. Cependant, il est quelquefois plus facile de prouver la contraposée d'une proposition que la proposition elle-même. Nous établissons le théorème :

```
>>> (P => Q) <=> (non Q) => (non P)
Théorème: pour toutes propositions Q, P,
   (P => Q) <=> (non Q) => (non P)
```

La démonstration par l'absurde consiste, pour démontrer $P \Rightarrow Q$, à supposer vraie l'hypothèse P et fausse la conclusion Q et à en déduire une contradiction — ce qui revient à dire qu'on en déduit la proposition « faux ». La validité de cette méthode de démonstration repose donc sur le théorème suivant :

```
>>> (P et non Q => faux) <=> (P => Q)
Théorème: pour toutes propositions Q, P,
   (P et non Q => faux) <=> (P => Q)
```

La réponse aux questions de l'introduction

Nous répondons maintenant aux interrogations du début de ce chapitre. Nous avions posé la question : « sachant que la phrase P est vraie et que la phrase Q est fausse, est-ce que la phrase obtenue en disant P *et* Q est une phrase vraie ? ». L'hypothèse « P est vraie et Q est fausse » se traduit simplement par P *et non* Q. Sous cette hypothèse, peut-on démontrer que P *et* Q est vraie ?

```
>>> (P et non Q) => (P et Q)
(P et non Q) => (P et Q) n'est pas un théorème,
car la proposition est fausse quand
P est vraie
Q est fausse
```

Non, la déduction est erronée. En revanche le contraire est exact :

```
>>> (P et non Q) => non (P et Q)
Théorème: pour toutes propositions Q, P,
   (P et non Q) => non (P et Q)
```

Examinons maintenant la validité du raisonnement complexe :

Puisque P est vraie et que Q est fausse,
P n'est donc pas une condition nécessaire pour que Q soit vraie

La première ligne ne pose pas de problème de traduction. Le raisonnement « puisque R, donc S » se traduit aisément : c'est une autre façon d'exprimer que l'implication « $R \Rightarrow S$ » est vraie. Le raisonnement se traduit donc par (P *et non* Q) $\Rightarrow \ldots$ En revanche, que signifie « condition nécessaire » ? P est une condition nécessaire pour Q si, dès que Q est vraie, alors P est elle aussi vraie. Autrement dit, si Q est vraie, P est nécessairement vraie. Cela signifie que l'on a $Q \Rightarrow P$. La phrase « P n'est pas une condition nécessaire pour que Q soit vraie » signifie donc simplement *non* ($Q \Rightarrow P$). En mettant les morceaux ensemble, on obtient :

```
>>> (P et non Q) => (non (Q => P))
(P et non Q) => (non (Q => P)) n'est pas un théorème,
car la proposition est fausse quand
P est vraie
Q est fausse
```

Le raisonnement n'était pas valide. On a confondu condition nécessaire et condition suffisante : une condition suffisante pour qu'une proposition Q soit vraie est une proposition P qui permet de déduire la proposition Q. Autrement dit, P est une condition suffisante pour que Q soit vraie si P suffit pour démontrer Q, c'est-à-dire si $P \Rightarrow Q$ est vraie. Sous les hypothèses du raisonnement, il est exact que P n'est pas une condition suffisante pour démontrer Q. En effet :

```
>>> (P et non Q) => (non (P => Q))
Théorème: pour toutes propositions Q, P,
   (P et non Q) => (non (P => Q))
```

Lois de De Morgan

Nous démontrons maintenant deux théorèmes, bien connus en logique sous le nom de lois de De Morgan, qui font le lien entre les connecteurs « et », « ou » et « non » :

```
>>> non (P et Q) <=> (non P) ou (non Q)
Théorème: pour toutes propositions Q, P,
   non (P et Q) <=> (non P) ou (non Q)
>>> non (P ou Q) <=> (non P) et (non Q)
Théorème: pour toutes propositions Q, P,
   non (P ou Q) <=> (non P) et (non Q)
```

Les lois de De Morgan sont quelquefois utiles en informatique. Par exemple, dans une alternative comme

```
if not (x <= 1 || x >= 2) then ... else ...
```

on simplifie la condition en employant les lois de De Morgan. En effet,

`not (x <= 1 || x >= 2)` signifie `(not (x <= 1)) && (not (x >= 2))`

c'est-à-dire `x > 1 && x < 2`. (Il faut savoir que le contraire de `<=` est `>`, celui de `>=` est `<` et réciproquement.)

Remarquez que nous pouvons démontrer :

```
>>> non ((non P) ou (non Q)) <=> P et Q
Théorème: pour toutes propositions Q, P,
   non ((non P) ou (non Q)) <=> P et Q
```

Cette propriété permet de réduire plus rapidement l'alternative ci-dessus. En remarquant que `x <= 1` est équivalent à `not (x > 1)` et que `x >= 2` est équivalent à `not (x < 2)`, on écrit la condition `not (x <= 1 || x >= 2)` sous la forme `not ((not (x > 1)) || (not (x < 2)))`. Il ne reste qu'à utiliser le théorème précédent avec $P =$ (`x > 1`) et $Q =$ (`x < 2`). On obtient alors P *et* Q, c'est-à-dire `x > 1 && x < 2`.

Si vous ne vous intéressez pas aux propriétés algébriques des connecteurs logiques, ou si vous n'êtes pas curieux de voir notre programme démontrer des propriétés « abstraites », vous pouvez passer à la section suivante.

Propriétés algébriques des connecteurs propositionnels

On établit en mathématiques que le « et » et le « ou » sont des opérations associatives et commutatives. Ces propriétés sont communes aux connecteurs propositionnels et aux opérations arithmétiques + et ×. Par exemple, pour l'addition, la commutativité signifie qu'on peut additionner les nombres dans n'importe quel ordre sans changer le résultat : $x + y = y + x$. L'associativité concerne les parenthèses ; elle indique simplement que la place des parenthèses ne change pas le résultat d'une addition, ce qu'on exprime par une formule qui montre que déplacer les parenthèses ne modifie pas le résultat du calcul : $(x + y) + z = x + (y + z)$. Le « ou » et le « et » vérifient ces propriétés :

```
>>> (P ou Q) <=> (Q ou P)
Théorème: pour toutes propositions Q, P, (P ou Q) <=> (Q ou P)
>>> ((P ou Q) ou R) <=> (P ou (Q ou R))
Théorème: pour toutes propositions R, Q, P,
   ((P ou Q) ou R) <=> (P ou (Q ou R))
```

L'implication est-elle aussi associative et commutative ?

```
>>> (P => Q) <=> (Q => P)
(P => Q) <=> (Q => P) n'est pas un théorème,
car la proposition est fausse quand
P est fausse
Q est vraie
>>> ((P => Q) => R) <=> (P => (Q => R))
((P => Q) => R) <=> (P => (Q => R)) n'est pas un théorème,
car la proposition est fausse quand
P est fausse
Q est vraie
R est fausse
```

Une propriété intéressante : une combinaison de « et » et de « ou » se comporte comme la multiplication et l'addition, on la « développe » de façon analogue. Il est bien connu que la multiplication est distributive par rapport à l'addition, ce qui permet d'écrire : $x \times (y + z) = x \times y + x \times z$. Notre programme prouve que le « et » est distributif par rapport au « ou », c'est-à-dire qu'on développe P *et* $(Q$ *ou* $R)$ comme si c'était $P \times (Q + R)$:

```
>>> (P et (Q ou R)) <=> (P et Q) ou (P et R)
Théorème: pour toutes propositions R, Q, P,
   (P et (Q ou R)) <=> (P et Q) ou (P et R)
```

Il démontre tout aussi facilement que le « ou » est distributif par rapport au « et » (ce résultat n'a pas d'analogue en arithmétique).

```
>>> (P ou (Q et R)) <=> (P ou Q) et (P ou R)
Théorème: pour toutes propositions R, Q, P,
   (P ou (Q et R)) <=> (P ou Q) et (P ou R)
```

Imaginez ce que serait une démonstration « à la main » de cette propriété avec des tables de vérité : on aurait huit lignes et huit colonnes ...

Finalement, l'équivalence est une *relation d'équivalence*, ce qui signifie simplement qu'elle introduit une certaine notion d'égalité. Il est évidemment heureux qu'il en soit ainsi : il serait vraiment dommage que l'équivalence ne soit pas une relation d'équivalence ! De plus, nous avons vu que l'équivalence définissait la notion de propositions logiquement identiques, c'est-à-dire égales au point de vue de la logique ; l'équivalence définit donc bien une notion d'égalité. Formellement, une relation d'équivalence est une relation réflexive, symétrique et transitive. L'égalité en mathématiques (le symbole =) a ces propriétés et ce sont les propriétés minimales qu'on doit exiger d'une relation pour qu'elle définisse l'idée de deux objets semblables. La principale difficulté pour expliciter ces propriétés provient de leur évidence même : nous y sommes tellement habitués et elles paraissent tellement évidentes que « ça va sans dire ». C'est exactement le but de la logique que d'écrire noir sur blanc ces évidences.

La réflexivité signifie qu'un objet est toujours égal à lui-même : $x = x$. La symétrie signifie que lorsqu'un objet est égal à un autre objet, l'autre objet est aussi égal au premier : si $x = y$ alors $y = x$. La transitivité se traduit par « deux objets égaux à un même troisième sont égaux entre eux », ou encore si $x = y$ et $y = z$ alors $x = z$. Le programme établit pour nous ces trois propriétés de l'équivalence logique :

```
>>> P <=> P
Théorème: pour toute proposition P, P <=> P
>>> (P <=> Q) => (Q <=> P)
Théorème: pour toutes propositions Q, P, (P <=> Q) => (Q <=> P)
>>> (P <=> Q) et (Q <=> R) => (P <=> R)
Théorème: pour toutes propositions R, Q, P,
   (P <=> Q) et (Q <=> R) => (P <=> R)
```

On utilise les deux théorèmes suivants pour simplifier les circuits logiques. Le premier permet d'éliminer deux connecteurs propositionnels :

```
>>> (P ou (P et Q)) <=> P
Théorème: pour toutes propositions Q, P, (P ou (P et Q)) <=> P
```

On supprime de même un connecteur et une négation grâce au théorème :

```
>>> (P ou (non P et Q)) <=> P ou Q
Théorème: pour toutes propositions Q, P,
   (P ou (non P et Q)) <=> P ou Q
```

Pour finir, un petit exercice (le premier exemple s'appelle la loi de Pierce) :

```
>>> ((P => Q) => P) => P
Théorème: pour toutes propositions Q, P, ((P => Q) => P) => P
>>> ((P => Q) => P) => Q
((P => Q) => P) => Q n'est pas un théorème,
car la proposition est fausse quand
```

```
P est vraie
Q est fausse
>>> (P <=> Q) <=> ((P ou Q) => (P et Q))
Théorème: pour toutes propositions Q, P,
   (P <=> Q) <=> ((P ou Q) => (P et Q))
```

L'auriez-vous deviné en réfléchissant au sens des connecteurs ?

12.8 Pour aller plus loin : l'analyseur lexical universel

Dans cette section, nous implémentons l'analyseur lexical « universel » utilisé pour lire les propositions. On rappelle l'interface de ce module :

Fichier `lexuniv.mli`

```
type lexème =
   | MC of string
   | Ident of string
   | Entier of int;;
value construire_analyseur:
     string list -> (char stream -> lexème stream);;
```

L'implémentation reprend de gros morceaux de l'analyseur lexical écrit pour le mini-Logo (chapitre 9, section 9.6), en les rendant un peu plus généraux. La principale nouveauté est l'introduction de *tables de hachage* pour décider rapidement si un identificateur est un mot-clé.

Les analyseurs lexicaux engendrés par la fonction `construire_analyseur` savent reconnaître les nombres entiers, les identificateurs et les mots-clés. Il serait facile d'ajouter les nombres flottants et les chaînes de caractères, mais cela n'est pas utile pour l'utilisation que nous en ferons.

Fonctions de lecture de base

Pour la lecture des entiers et des identificateurs, nous réutilisons les fonctions `lire_entier` et `lire_mot` du mini-Logo.

Fichier `lexuniv.ml`

```
let rec lire_entier accumulateur flux =
  match flux with
  | [< '(‘0‘..‘9‘ as c) >] ->
      lire_entier (10 * accumulateur + int_of_char c - 48) flux
  | [< >] ->
      accumulateur;;

let tampon = make_string 16 ‘-‘;;

let rec lire_mot position flux =
  match flux with
  | [< '(‘A‘..‘Z‘ | ‘a‘..‘z‘ | ‘0‘..‘9‘ | ‘_‘ | ‘’‘ |
         ‘é‘|‘à‘|‘è‘|‘ù‘|‘â‘|‘ê‘|‘î‘|‘ô‘|‘û‘|‘ë‘|‘ï‘|‘ü‘|‘ç‘|
         ‘É‘|‘À‘|‘È‘|‘Ù‘|‘Â‘|‘Ê‘|‘Î‘|‘Ô‘|‘Û‘|‘Ë‘|‘Ï‘|‘Ü‘|‘Ç‘
         as c) >] ->
```

```
      if position < string_length tampon then
        tampon.[position] <- c;
      lire_mot (position + 1) flux
  | [< >] ->
      sub_string tampon 0 (min position (string_length tampon));;
```

Sur le modèle de `lire_mot`, nous ajoutons une fonction `lire_symbole` qui reconnaît les suites de caractères spéciaux, comme `**` ou `<=>`. Ces suites de caractères spéciaux sont soit des mots-clés soit des identificateurs, exactement comme les suites de lettres et de chiffres qui constituent un mot.

Fichier `lexuniv.ml`

```
let rec lire_symbole position flux =
  match flux with
  | [< '('!'|'$'|'%'|'&'|'*'|'+'|'-'|'.'|'/'|':'|
         ';'|'<'|'='|'>'|'?'|'@'|'^'|'|'|'~' as c) >] ->
      if position < string_length tampon then
        tampon.[position] <- c;
      lire_symbole (position + 1) flux
  | [< >] ->
      sub_string tampon 0 (min position (string_length tampon));;
```

L'analyseur lexical autorise des commentaires dans le texte d'entrée, sous une forme très simple : tout ce qui suit un caractère `#` est ignoré, jusqu'à la fin de la ligne. La fonction `lire_commentaire` se charge de sauter tous les caractères du flux d'entrée jusqu'au prochain caractère de fin de ligne.

Fichier `lexuniv.ml`

```
let rec lire_commentaire flux =
  match flux with
  | [< '('\n' >] -> ()
  | [< 'c >] -> lire_commentaire flux;;
```

Recherche des mots-clés par hachage

Ayant reconnu un mot ou un symbole, il reste à déterminer s'il appartient à la liste des mots-clés. Ce test sera fait une fois pour chaque mot ou symbole du flux d'entrée, donc un très grand nombre de fois ; il est par conséquent crucial que ce test « aille vite ». Au lieu d'une simple recherche linéaire dans la liste des mots-clés, nous employons une technique plus subtile, connue sous le nom de *hachage*, et la structure de données correspondante, les *tables de hachage*. La technique du hachage est décrite en détail à la prochaine section. Pour l'instant, nous allons juste expliquer le comportement d'une table de hachage. Une table de hachage est une table d'association physiquement modifiable : elle enregistre des associations de certaines clés avec certaines données ; on entre de nouvelles associations, ou l'on en retire d'anciennes en modifiant physiquement la table. Voici une partie de l'interface du module `hashtbl` de la bibliothèque standard, qui implémente les tables de hachage :

```
type ('a, 'b) t;;
value new: int -> ('a, 'b) t
```

```
        and add: ('a, 'b) t -> 'a -> 'b -> unit
        and find: ('a, 'b) t -> 'a -> 'b;;
```

Le type `('a, 'b) hashtbl__t` est le type des tables de hachage associant des clés de type `'a` avec des données de type `'b`. La fonction `hashtbl__new` renvoie une nouvelle table de hachage, initialement vide, c'est-à-dire ne contenant aucune liaison. (Le paramètre entier de `hashtbl__new` est une indication de la taille de la table ; nous verrons son rôle dans la prochaine section.) La fonction `hashtbl__find` effectue une recherche dans une table de hachage : `hashtbl__find t c` renvoie la donnée à laquelle la clé `c` est liée dans la table `t`, ou déclenche l'exception `Not_found` si la clé `c` n'est pas liée. La fonction `hashtbl__add` enregistre une liaison dans une table de hachage : `hashtbl__add t c d` place dans la table `t` une liaison de la clé `c` à la donnée `d`. Si la clé `c` était déjà liée à une donnée, la nouvelle liaison cache l'ancienne.

Pour résumer, disons que les tables de hachage se comportent exactement comme des listes d'association physiquement modifiables. Poussant cette intuition, on peut très bien faire une implémentation correcte du module `hashtbl` à l'aide de listes d'association, sans employer la technique du hachage :

```
type ('a, 'b) t == ('a * 'b) list ref;;
let new n = ref [];;
let add t c d = t := (c, d) :: !t;;
let find t c = assoc c !t;;
```

La vraie implémentation de `hashtbl`, esquissée dans la prochaine section, est bien plus complexe : elle utilise du hachage pour accélérer considérablement l'opération `find`. Cependant, son comportement est exactement le même que celui de l'implémentation naïve donnée ci-dessus. Retenons donc qu'une table de hachage se comporte comme une liste d'association, à ceci près que les opérations de recherche sont beaucoup plus efficaces.

Nous utiliserons donc une table de hachage pour stocker l'ensemble des mots-clés d'un analyseur lexical. La table associe aux mots-clés eux-mêmes (des chaînes de caractères) les lexèmes correspondants. Elle est donc du type `(string, lexème) hashtbl__t`. Pour déterminer si un mot trouvé dans le flux d'entrée est un mot-clé ou un simple identificateur, on interroge tout simplement la table des mots-clés avec `hashtbl__find`.

Fichier `lexuniv.ml`

```
let mc_ou_ident table_des_mots_clés ident =
    try hashtbl__find table_des_mots_clés ident
    with Not_found -> Ident ident;;
```

Une variante de `mc_ou_ident` nous sera utile pour reconnaître les mots-clés mono-caractères, par exemple les parenthèses.

Fichier `lexuniv.ml`

```
let mc_ou_erreur table_des_mots_clés caractère =
    let ident = make_string 1 caractère in
    try hashtbl__find table_des_mots_clés ident
    with Not_found -> raise Parse_error;;
```

Reconnaissance d'un lexème

La lecture des lexèmes consiste tout d'abord à passer les blancs et les commentaires, puis à reconnaître un identificateur, un mot-clé ou un nombre entier (éventuellement négatif, donc précédé du signe « - »). Les symboles mono-caractères comme les parenthèses `()[]{}` ou bien les caractères non imprimables doivent être déclarés comme mots-clés, sinon ils produisent une erreur.

Fichier `lexuniv.ml`

```
let rec lire_lexème table flux =
  match flux with
  | [< '(' ' | '\n' | '\r' | '\t') >] ->
      lire_lexème table flux
  | [< ''#' >] ->
      lire_commentaire flux; lire_lexème table flux
  | [< '('A'..'Z' | 'a'..'z' |
         'é'|'à'|'è'|'ù'|'â'|'ê'|'î'|'ô'|'û'|'ë'|'ï'|'ü'|'ç'|
         'É'|'À'|'È'|'Ù'|'Â'|'Ê'|'Î'|'Ô'|'Û'|'Ë'|'Ï'|'Ü'|'Ç'
        as c) >] ->
      tampon.[0] <- c;
      mc_ou_ident table (lire_mot 1 flux)
  | [< '('!'|'$'|'%'|'&'|'*'|'+'|'.'|'/'|':'|';'|
         '<'|'='|'>'|'?'|'@'|'^'|'|'|'~' as c) >] ->
      tampon.[0] <- c;
      mc_ou_ident table (lire_symbole 1 flux)
  | [< '('0'..'9' as c) >] ->
      Entier(lire_entier (int_of_char c - 48) flux)
  | [< ''-' >] ->
      begin match flux with
      | [< '('0'..'9' as c) >] ->
          Entier(- (lire_entier  (int_of_char c - 48) flux))
      | [< >] ->
          tampon.[0] <- '-';
          mc_ou_ident table (lire_symbole 1 flux)
      end
  | [< 'c >] ->
      mc_ou_erreur table c;;
```

Génération de l'analyseur

Comme pour le mini-Logo, on construit le flux des lexèmes par appels répétés à `lire_lexème`.

Fichier `lexuniv.ml`

```
let rec analyseur table flux =
    stream_from (function () ->
     match flux with
     | [< (lire_lexème table) lexème >] -> lexème
     | [< >] -> raise Parse_failure);;
```

Finalement, la génération d'un analyseur lexical consiste simplement à construire sa table des mots-clés, puis à appliquer partiellement l'analyseur générique à cette table. Le résultat de l'application partielle est la fonction des flux de caractères vers les flux de lexèmes désirée.

Fichier `lexuniv.ml`

```
let construire_analyseur mots_clés =
    let table_des_mots_clés = hashtbl__new 17 in
    do_list
      (function mot -> hashtbl__add table_des_mots_clés mot (MC mot))
      mots_clés;
    analyseur table_des_mots_clés;;
```

12.9 Pour aller encore plus loin : le hachage

L'idée du hachage vient de la constatation que la recherche d'un objet dans une liste d'association se révèle coûteuse si l'on doit faire beaucoup de recherches, en particulier si ces recherches sont la plupart du temps infructueuses. En effet pour trouver un objet dans une liste d'association, il faut en moyenne parcourir la moitié de la liste, si l'on suppose qu'on recherche des clés en moyenne disposées au hasard dans la liste. Pour constater que la clé est absente de la liste, c'est pire : il faut parcourir toute la liste. (C'est le cas le plus fréquent dans l'exemple de l'analyse lexicale.)

Le seul moyen d'accélérer la recherche d'une clé dans un ensemble est d'éliminer très rapidement un grand nombre de tests en prouvant très vite qu'ils sont voués à l'échec. En particulier, on ira beaucoup plus vite si l'on est capable de restreindre la recherche exhaustive de la clé à un ensemble beaucoup plus petit que celui de départ. C'est toujours le principe de « diviser pour régner » qui prévaut.

Le hachage consiste donc à fractionner un gros ensemble de clés en sous-ensembles cohérents et à ne chercher une clé que dans le petit sous-ensemble qui la concerne. La méthode suppose donc qu'on dispose d'un moyen très rapide de déterminer le sous-ensemble auquel une clé est susceptible d'appartenir. Les sous-ensembles sont par exemple de simples listes ; on les regroupe en un tableau, afin de pouvoir accéder directement à n'importe quel sous-ensemble. On représente donc le sous-ensemble auquel appartient une clé par un simple numéro, l'indice du sous-ensemble dans le tableau des sous-ensembles. La fonction qui détermine ce numéro s'appelle justement la fonction de hachage.

En termes savants, on dit qu'on partitionne les clés en classes d'équivalence modulo la fonction de hachage. En termes simples, on range les clés dans le même sous-ensemble quand elles ont la même image par la fonction de hachage. Prenons un exemple très simple : supposons que les clés soient des nombres entiers. Comment partager rapidement ces entiers en dix sous-ensembles ? Il suffit de regarder leur dernier chiffre. Si l'on veut les partager en deux sous-ensembles, on considère leur parité (pair ou impair). De façon générale, une manière de les partager en n sous-ensembles est de calculer le reste de leur division par n. À titre démonstratif, nous prenons dix sous-ensembles. Notre fonction de hachage est donc :

```
# let hache clé = clé mod 10;;
```

```
hache : int -> int = <fun>
```

Maintenant, nous voulons associer des informations à nos entiers, par exemple des chaînes de caractères. C'est notamment le cas si nous voulons représenter un annuaire « à l'envers » : à partir d'un numéro de téléphone, nous désirons retrouver le nom du correspondant. Sous forme de liste d'association, cela donne :

```
# let liste_d'association =
      [11, "police"; 16, "pompiers"; 0139635511, "standard";
       0139635198, "Pierre"; 0139635202, "Xavier"; 7234864, "Xavier";
       0139635570, "Nelly"; 3613, "Télétel 1"; 3615, "Télétel 3" ];;
```

Sous forme de table de hachage, nous divisons cette liste en dix listes d'association, suivant le dernier chiffre du numéro :

```
# let table_des_sous_ensembles =
      [| (* 0 *) [0139635570, "Nelly"];
         (* 1 *) [11, "police"; 0139635511, "standard"];
         (* 2 *) [0139635202, "Xavier"];
         (* 3 *) [3613, "Télétel 1"];
         (* 4 *) [7234864, "Xavier"];
         (* 5 *) [3615, "Télétel 3"];
         (* 6 *) [16, "pompiers"];
         (* 7 *) [];
         (* 8 *) [0139635198, "Pierre"];
         (* 9 *) []
      |];;
```

Pour trouver le sous-ensemble dans lequel chercher une clé, on cherche son numéro en « hachant » la clé puis on extrait du tableau le sous-ensemble concerné. Pour chercher l'associé d'une clé on utilise simplement `assoc` sur le sous-ensemble correspondant à la clé.

```
# let sous_ensemble_de clé =
      let numéro_du_sous_ensemble = hache clé in
      table_des_sous_ensembles.(numéro_du_sous_ensemble);;
sous_ensemble_de : int -> (int * string) list = <fun>
# let associé_de clé = assoc clé (sous_ensemble_de clé);;
associé_de : int -> string = <fun>
# associé_de 3615;;
- : string = "Télétel 3"
# associé_de 911;;
Exception non rattrapée: Not_found
```

Chaque appel à `associé_de` finit donc par appeler la fonction `assoc`, mais sur des listes d'association beaucoup plus petites que la liste représentant tout l'annuaire : un et deux éléments, respectivement, au lieu de neuf. Dans certains cas, on tombe même immédiatement sur un sous-ensemble vide, par exemple si on cherche un numéro se terminant par 7, ce qui fait que la recherche est quasi immédiate. Dans tous les cas, on restreint nettement l'espace de recherche.

Il est facile de construire les sous-ensembles automatiquement. On part d'une table où tous les sous-ensembles sont vides.

```
# let table_des_sous_ensembles =
    (make_vect 10 [] : (int * string) list vect);;
```

Puis on range chaque paire (clé, valeur) à mémoriser dans le sous-ensemble correspondant à la valeur de hachage de la clé.

```
# let ajoute_une_clé ((clé, valeur) as clé_valeur) =
      let numéro_du_sous_ensemble = hache clé in
      table_des_sous_ensembles.(numéro_du_sous_ensemble) <-
        clé_valeur ::
        table_des_sous_ensembles.(numéro_du_sous_ensemble);;
ajoute_une_clé : int * string -> unit = <fun>
# do_list ajoute_une_clé liste_d'association;;
- : unit = ()
# table_des_sous_ensembles;;
- : (int * string) list vect =
 [|[139635570, "Nelly"]; [139635511, "standard"; 11, "police"];
   [139635202, "Xavier"]; [3613, "Télétel 1"]; [7234864, "Xavier"];
   [3615, "Télétel 3"]; [16, "pompiers"]; []; [139635198, "Pierre"];
   []|]
```

Le hachage n'est pas restreint aux clés de type entier. On peut l'appliquer à n'importe quel type de clés, pourvu qu'on sache associer rapidement un entier à une clé. On définit alors la fonction de hachage comme étant l'entier associé à la clé, modulo la taille de la table de hachage. La transformation de la clé en entier n'a pas besoin d'être « exacte », en ce sens que deux clés différentes ont sans problème le même entier associé. Pour obtenir une bonne répartition des clés dans les sous-ensembles, il faut quand même s'efforcer d'éviter autant que possible cette situation. Dans le cas particulier où les clés sont des chaînes de caractères, cas d'une grande importance pratique, on a proposé un certain nombre de « recettes » pour associer rapidement un entier à une chaîne, avec de bonnes propriétés de répartition. Voici un exemple simple de fonction de hachage sur les chaînes :

```
# let hache_chaîne taille_table c =
      let res = ref 0 in
      for i = 0 to string_length c - 1 do
        res :=
          (int_of_char c.[i] + !res * 128) mod taille_table
      done;
      !res;;
hache_chaîne : int -> string -> int = <fun>
```

L'idée est de faire intervenir dans le résultat final la valeur de chacun des caractères de la chaîne, pour assurer une bonne dispersion des résultats. Nous n'essaierons pas de justifier la formule ci-dessus dans tous ses détails (pourquoi 128, etc.).

Nous savons donc hacher des entiers et des chaînes. Le système Caml va beaucoup plus loin que cela : il fournit une fonction de bibliothèque capable d'associer un entier à n'importe quelle valeur Caml, quel que soit son type. Il s'agit de la fonction `hashtbl__hash`, de type `'a -> int`. Cette fonction est raisonnablement rapide et produit des résultats assez bien répartis. Au-dessus de cette fonction, il est facile de définir une fonction de hachage qui opère sur tous les types de clés :

```
# let hache taille_table clé =
    hashtbl__hash clé mod taille_table;;
hache : int -> 'a -> int = <fun>
```

Ensuite, on construit facilement un type `t` et des opérations `new`, `add` et `find` comparables à ceux du module `hashtbl` :

```
# type ('a, 'b) t == ('a * 'b) list vect;;
Le type t est défini.
# let new taille_table =
      make_vect taille_table [];;
new : int -> 'a list vect = <fun>
# let add table clé donnée =
      let index = hache (vect_length table) clé in
      table.(index) <- (clé, donnée) :: table.(index);;
add : ('a * 'b) list vect -> 'a -> 'b -> unit = <fun>
# let find table clé =
      let index = hache (vect_length table) clé in
      assoc clé table.(index);;
find : ('a * 'b) list vect -> 'a -> 'b = <fun>
```

L'implémentation du module `hashtbl` fournie par la bibliothèque standard s'appuie elle aussi sur la fonction `hash` polymorphe, mais est plus complexe que l'implémentation donnée ci-dessus. En particulier, elle sait agrandir dynamiquement la table quand les sous-ensembles menacent de devenir trop gros, ce qui garantit de bonnes performances même sur de très grosses tables.

13
Compression de fichiers

Où l'on fait passer un chameau par le chas d'une aiguille.

Dans ce chapitre, nous programmerons une commande de compression de fichiers. La compression consiste à transformer des fichiers pour qu'ils occupent moins de place ; l'opération inverse, la décompression, reconstruit les fichiers de départ à partir des fichiers transformés. Ce sera l'occasion d'introduire quelques algorithmes classiques, en particulier deux exemples intéressants d'utilisation des arbres binaires, parmi bien d'autres. Nous aurons également besoin de faire des entrées-sorties bit par bit, et donc de manipuler les entiers au niveau du bit.

13.1 La compression de données

La plupart des fichiers stockés dans les ordinateurs contiennent un certain degré de redondance. Très souvent, si l'on code différemment les données qu'ils contiennent, on réduit considérablement leur taille, sans perte d'information, si l'on suppose évidemment que le processus de recodage est réversible, et qu'il permet donc de retrouver les fichiers d'origine à tout instant. C'est ce recodage qu'on appelle compression des données.

Les procédés de compression et de décompression de données sont de plus en plus employés dans les environnements informatiques : en premier lieu dans des programmes utilitaires spécialisés comme `gzip`, `stuffit` ou `pkzip`, qui souvent combinent compression et archivage (regroupement d'une hiérarchie de fichiers en un seul fichier) ; mais aussi dans certains pilotes de disques, qui compressent "au vol" les données avant de les écrire sur le disque, augmentant ainsi la capacité apparente de ce dernier ; et même dans l'électronique des modems, qui compressent "au vol" (en temps réel) les données transmises sur la ligne téléphonique, augmentant ainsi le débit des transmissions.

En guise d'exemple très simple d'algorithme de compression, mentionnons la méthode dite *run-length encoding*, qui consiste à représenter toute séquence de n fois le même octet c par un code spécial signifiant "répétition", suivi de l'octet c, suivi du nombre de répétitions n. Ce codage est plus compact que l'original dès que n est plus grand que 4. Il est intéressant sur certains types de fichiers, comme les sorties

pour l'imprimante en informatique de gestion, qui comportent de longues séquences de blancs (pour aligner) et de tirets (pour tracer des traits). Cependant, il est à peu près inefficace sur d'autres types de fichiers, comme les textes français ou les fichiers de code exécutable.

L'algorithme de compression que nous utilisons dans ce chapitre, l'algorithme de Huffman, est plus compliqué, mais plus efficace car il n'est pas limité à une classe particulière de données. Sur des fichiers de texte français, il atteint une réduction de taille d'environ 35 % en moyenne. Les meilleurs programmes de compression dépassent 60 %, mais ils utilisent des algorithmes encore plus complexes.

13.2 Plan du programme

Nous allons programmer la commande `compr` qui compresse les données des fichiers qu'elle traite. Les fichiers compressés sont renommés en ajoutant le suffixe `.cpr` à leur nom. Quand on l'appelle avec l'option `-d`, la commande `compr` décompresse les fichiers qu'on lui donne en argument.

Nous commençons par une fonction commune à la compression et à la décompression, qui se charge d'ouvrir les fichiers et d'afficher les erreurs d'entrée-sortie. Cette tâche conceptuellement simple est en pratique fort encombrée par la récupération et l'affichage des erreurs ; c'est le prix à payer pour obtenir des programmes qui réagissent bien face aux situations exceptionnelles.

Fichier `compr.ml`

```
#open "sys";;
exception Erreur;;

let traite_fichier traitement nom_entrée nom_sortie =
  let entrée =
    try open_in_bin nom_entrée
    with Sys_error message ->
      prerr_endline ("Erreur à l'ouverture de " ^ nom_entrée
                     ^ " : " ^ message);
      raise Erreur in
  let sortie =
    try open_out_bin nom_sortie
    with Sys_error message ->
      close_in entrée;
      prerr_endline ("Erreur à la création de " ^ nom_sortie
                     ^ " : " ^ message);
      raise Erreur in
  try
    traitement entrée sortie;
    close_in entrée; close_out sortie; remove nom_entrée
  with Sys_error message ->
    close_in entrée; close_out sortie; remove nom_sortie;
    prerr_endline ("Erreur pendant le traitement de "
                   ^ nom_entrée ^ " : " ^ message);
    raise Erreur;;
```

La fonction commence par ouvrir un canal d'entrée et un canal de sortie sur les fichiers indiqués, au moyen des fonctions `open_in_bin` et `open_out_bin`. Les fonctions de bibliothèque `open_in_bin` et `open_out_bin` ouvrent les canaux en mode "binaire", garantissant que les caractères lus ou écrits sur le canal sont exactement ceux que contiennent le fichier. Au contraire, les fonctions `open_in` et `open_out` ouvrent les canaux en mode "texte" ; sur certaines implémentations de Caml Light, il se produit alors des traductions (en particulier sur les caractères de fin de lignes) au moment de la lecture et de l'écriture. Les fichiers à compresser ne contiennent pas forcément du texte ; il est donc nécessaire d'opérer en mode "binaire" pour être certain de retrouver les fichiers à l'identique après une compression suivie d'une décompression. La partie difficile du travail, c'est-à-dire la compression proprement dite, est assurée par la fonction `traitement` passée en argument à `traite_fichier`. Cette fonction reçoit un canal ouvert sur l'entrée et un canal ouvert sur la sortie. Elle est censée compresser ou décompresser son entrée sur sa sortie. Lorsque la compression s'achève sans erreur, on ferme les canaux d'entrée et de sortie (fonctions `close_in` et `close_out`) et on efface le fichier d'entrée (fonction `remove` du module `sys`). Aux yeux de l'utilisateur, tout se passe comme si on avait remplacé le fichier d'entrée par le fichier de sortie.

Toutes les fonctions d'entrée-sortie déclenchent l'exception `Sys_error` (du module `sys`) quand une erreur se produit, avec un message explicatif en argument de l'exception. On intercepte donc cette exception, et on affiche un message sur la sortie d'erreur standard du processus. La fonction `prerr_endline` écrit une chaîne de caractères suivie d'un retour à la ligne sur la sortie d'erreur standard. En cas d'erreur, on détruit le fichier de sortie s'il a déjà été créé, et on déclenche l'exception `Erreur`. On prend bien soin de fermer les canaux quand on n'en a plus besoin, y compris en cas d'erreurs. Ce n'est pas uniquement par souci d'élégance : les systèmes d'exploitation limitent le nombre de canaux d'entrées-sorties simultanément ouverts. Si on oublie de fermer les canaux inutilisés, on se trouve vite en situation de pénurie de canaux.

Nous allons maintenant utiliser deux fois la fonction `traite_fichier`, pour définir les fonctions de compression et de décompression d'un fichier.

Fichier `compr.ml`

```
let compresse_fichier nom_fichier =
  traite_fichier huffman__compresse
                 nom_fichier (nom_fichier ^ ".cpr");;

let décompresse_fichier nom_fichier =
  let longueur = string_length nom_fichier in
  if longueur < 4
  || sub_string nom_fichier (longueur - 4) 4 <> ".cpr" then
    let nom_entrée = nom_fichier ^ ".cpr"
    and nom_sortie = nom_fichier in
    traite_fichier huffman__décompresse nom_entrée nom_sortie
  else
    let nom_entrée = nom_fichier
    and nom_sortie = sub_string nom_fichier 0 (longueur - 4) in
    traite_fichier huffman__décompresse nom_entrée nom_sortie;;
```

Dans le cas de la décompression, le nom du fichier compressé peut être donné avec

ou sans l'extension `.cpr`. Si le nom fourni ne se termine pas par l'extension `.cpr`, c'est en fait le nom du fichier de sortie ; on lui ajoute `.cpr` pour obtenir le vrai nom du fichier d'entrée. Si le nom fourni se termine par l'extension `.cpr`, on l'enlève (par un `sub_string` bien calculé) pour obtenir le nom du fichier de sortie.

Les deux fonctions `huffman__compresse` et `huffman__décompresse` proviennent du module `huffman`, que nous étudierons en détail dans la prochaine section. Pour l'instant, contentons-nous de l'interface de ce module :

Fichier `huffman.mli`

```
value compresse : in_channel -> out_channel -> unit
  and décompresse : in_channel -> out_channel -> unit;;
```

Le point d'entrée dans le programme reconnaît l'option `-d` et applique à bon escient les fonctions `compresse_fichier` ou `décompresse_fichier` à chaque argument fourni sur la ligne de commande. Les arguments donnés à une commande sont accessibles dans le tableau de chaînes de caractères `command_line`, du module de bibliothèque `sys`. L'élément d'indice zéro contient le nom d'appel de la commande ; les éléments suivants, les arguments de la commande.

Fichier `compr.ml`

```
if sys__interactive then () else
  begin
    let erreur = ref false in
    if vect_length command_line >= 2 & command_line.(1) = "-d" then
      for i = 2 to vect_length command_line - 1 do
        try décompresse_fichier command_line.(i)
        with Erreur -> erreur := true
      done
    else
      for i = 1 to vect_length command_line - 1 do
        try compresse_fichier command_line.(i)
        with Erreur -> erreur := true
      done;
    exit (if !erreur then 2 else 0)
  end;;
```

Les deux boucles récupèrent l'exception `Erreur` pour passer au prochain argument de la ligne de commande en cas d'erreur. On positionne cependant le drapeau `erreur`, pour pouvoir renvoyer au système d'exploitation un code d'erreur approprié : le code zéro si aucune erreur ne s'est produite, un code non nul sinon.

Il est maintenant temps de passer aux choses sérieuses : l'implémentation des algorithmes de compression et de décompression.

13.3 L'algorithme de Huffman

L'algorithme de compression de Huffman repose sur l'observation que certains caractères apparaissent plus fréquemment que d'autres dans les fichiers. Par exemple, dans un fichier de texte, `e` apparaît plus souvent que `z` et l'espace apparaît plus souvent que le caractère tilde. Au lieu de coder chaque caractère sur huit bits quelle que soit

sa fréquence, nous allons attribuer des codes de longueur variable aux caractères, en faisant en sorte que les caractères les plus fréquents reçoivent des codes courts (moins de huit bits) et les caractères les plus rares des codes longs (éventuellement plus de huit bits). Par exemple, le codage suivant conviendrait pour du texte écrit en français :

espace	110	t	1000
e	010	i	0010
s	1001	r	0001
a	0011	u	11101
n	0111	l	10111

pour les lettres les plus fréquentes et ainsi de suite jusqu'aux lettres les plus rares :

X	11100110111100111
Y	011010001010000010
Z	011010001010000001

Compression

Compresser un fichier consiste à remplacer chaque octet du fichier par la suite de bits qui l'encode, puis à écrire octet par octet la suite de bits obtenus. Avec le codage ci-dessus, le mot `utile` suivi d'un espace est transformé en la suite de bits 11101.1000.0010.10111.010.110, c'est-à-dire en les trois octets 55 (11101100), 168 (00010101) et 107 (11010110). (On a choisi arbitrairement de lire les nombres binaires avec le bit de poids faible à gauche et le bit de poids fort à droite.) Le texte compressé occupe trois octets, au lieu de six pour le texte d'origine.

Pour éviter les problèmes qui se posent quand le dernier octet du fichier compressé n'est pas entièrement rempli, on convient de terminer tous les fichiers compressés par un code spécial de fin. Ce code n'apparaissant qu'une fois pour tout le fichier, il peut être choisi assez long.

Voici maintenant la fonction qui compresse un fichier (correspondant au descripteur de fichier `entrée`) et écrit le résultat sur un autre fichier (correspondant au descripteur de fichier `sortie`).

Fichier `huffman.ml`

```
type table_de_codage =
  { caractère: int list vect;
    mutable fin: int list };;

let encode entrée sortie codage =
  esbit__initialise ();
  try
    while true do
      let c = input_char entrée in
      do_list (esbit__écrire_bit sortie)
              codage.caractère.(int_of_char c)
    done
  with End_of_file ->           (* fin du fichier d'entrée *)
    do_list (esbit__écrire_bit sortie) codage.fin;
    esbit__finir sortie;;
```

Le codage employé est représenté par un enregistrement du type `table_de_codage`. La partie `caractère` est un tableau de 256 codes (un pour chaque octet). La partie `fin` est le code signalant la fin du fichier compressé. Les codes sont représentés par des listes d'entiers, `0` ou `1`.

La lecture du fichier d'entrée se fait avec la fonction `input_char` de la bibliothèque standard. Cette fonction renvoie le prochain caractère du canal d'entrée passé en argument. Elle déclenche l'exception `End_of_file` lorsque la fin du fichier est atteinte. La manière habituelle de lire tous les caractères d'un fichier est de faire `input_char` à l'intérieur d'une boucle infinie `while true do ... done`. L'exception `End_of_file` de fin de fichier fait sortir de la boucle ; elle doit être récupérée par une construction `try ... with`.

La fonction `encode` fait appel à un module `esbit` (pour « entrées-sorties bit à bit »), qui permet d'écrire sur un fichier non pas octet par octet comme les fonctions d'entrée-sortie usuelles, mais bit par bit. Nous implémenterons ce module plus tard. Pour l'instant, voici son l'interface :

Fichier `esbit.mli`

```
value initialise : unit -> unit
  and écrire_bit : out_channel -> int -> unit
  and lire_bit : in_channel -> int
  and finir : out_channel -> unit;;
```

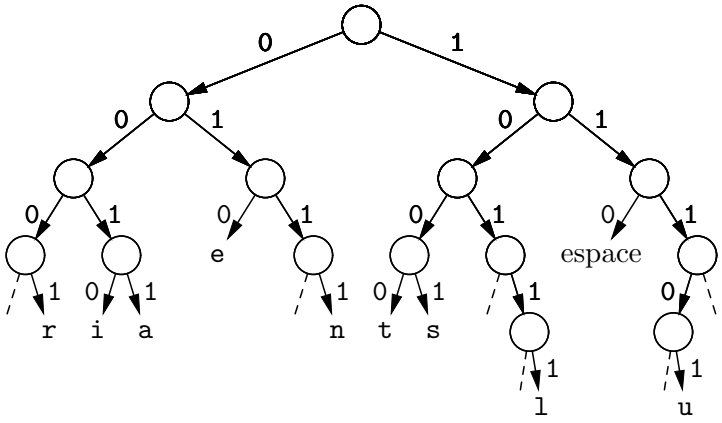
Les fonctions `écrire_bit` et `lire_bit` permettent d'écrire ou de lire un bit, représenté par les entiers 0 ou 1. La fonction `initialise` doit être appelée avant le premier appel à `écrire_bit` ou `lire_bit`. La fonction `finir` doit être appelée après le dernier appel à `écrire_bit`, pour effectuer les éventuelles écritures en attente. La fonction `encode` montre un bel exemple d'application partielle : on itère, avec la fonctionnelle `do_list`, la fonction `(esbit__écrire_bit sortie)`, obtenue par application partielle de `esbit__écrire_bit` à la sortie courante ; cette fonction n'est donc calculée qu'une fois, avant de lancer l'itération.

Décompression

La décompression se heurte à un petit problème : dans la suite de bits produite par l'algorithme de compression, rien ne marque les séparations entre les codes des différents octets. Il est cependant possible de reconstituer le texte d'origine, à condition que le codage employé ne soit pas ambigu : aucun code d'un caractère ne doit être préfixe du code d'un autre caractère. Supposons par exemple que le caractère `o` a pour code `0101`, qui a pour préfixe `010`, le code de `e`. Alors la suite de bits `01010111` représente aussi bien `el` que `on`. Au contraire, si aucun code n'est préfixe d'un autre, il y a une et une seule manière de découper la suite de bits contenue dans le fichier compressé.

Pour décrire plus précisément le processus de décodage, il est commode de représenter le codage sous la forme d'un arbre de Huffman. C'est est un arbre binaire dont les feuilles sont des caractères. Tout codage non ambigu est représenté par un arbre de Huffman, de la manière suivante : le code de chaque caractère est le chemin qui mène de la racine de l'arbre à la feuille portant ce caractère, avec la convention que `0` signifie « prendre la branche de gauche » et `1` signifie « prendre la branche de

droite ». Par exemple, voici l'arbre de Huffman pour le codage donné page 241 (nous représentons les arbres avec la racine en haut et les feuilles en bas, comme c'est l'usage en informatique) :



La décompression est très simple quand on dispose de l'arbre de Huffman du codage. On part de la racine de l'arbre. Si on est sur un nœud, on lit le prochain bit du fichier compressé et on va à gauche si c'est zéro et à droite si c'est un. Quand on aboutit sur une feuille, on émet la lettre correspondante et on repart de la racine de l'arbre. Cet algorithme s'écrit sans difficultés en Caml. (On a introduit un second type de feuille, le constructeur `Fin`, pour représenter le code de fin de fichier.)

Fichier `huffman.ml`

```
type arbre_de_huffman =
   | Lettre of char
   | Fin
   | Noeud of arbre_de_huffman * arbre_de_huffman;;

let décode entrée sortie arbre =
  esbit__initialise ();
  let rec parcours = function
  | Fin -> ()
  | Lettre c ->
      output_char sortie c; parcours arbre
  | Noeud(gauche, droite) ->
      if esbit__lire_bit entrée = 0
      then parcours gauche
      else parcours droite in
  parcours arbre;;
```

Détermination d'un codage adapté

On peut utiliser les fonctions `encode` et `décode` avec un codage de Huffman fixé, déterminé une fois pour toutes à partir des fréquences moyennes d'apparition des caractères dans les textes français (par exemple). Cependant, la compression risque d'être peu efficace sur d'autres types de textes (des programmes Caml, par exemple) ou sur des fichiers contenant autre chose que du texte (des images numérisées, par exemple). Pour plus de généralité, il vaut mieux déterminer les fréquences des caractères dans

le fichier à compresser, puis construire un codage de Huffman adapté à cette distribution de fréquence particulière. Bien entendu, le décompresseur ne peut pas deviner le codage que le compresseur a utilisé ; le compresseur écrit donc ce codage en tête du fichier compressé.

Calculer les fréquences d'apparition (c'est-à-dire le nombre d'occurrences) des caractères dans un fichier ne présente aucune difficulté.

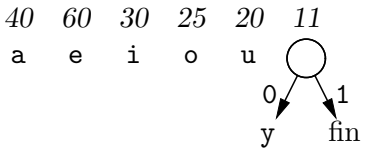
Fichier `huffman.ml`

```
let fréquences entrée =
  let fr = make_vect 256 0 in
  begin try
    while true do
      let c = int_of_char(input_char entrée) in fr.(c) <- fr.(c) + 1
    done
  with End_of_file -> ()
  end;
  fr;;
```

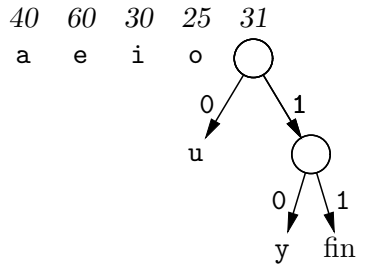
La fonction ci-dessus renvoie un tableau de 256 entiers qui donne le nombre d'occurrences de chaque caractère dans le fichier `entrée`. Déterminer un codage adapté à ce tableau de fréquences est plus difficile. Voici un algorithme qui construit un arbre de Huffman petit à petit, à partir d'un ensemble de feuilles, une par caractère apparaissant dans le fichier, plus une pour la fin du fichier. Chaque feuille est annotée par la fréquence d'apparition du caractère correspondant :

40	*60*	*30*	*25*	*20*	*10*	*1*
a	e	i	o	u	y	fin

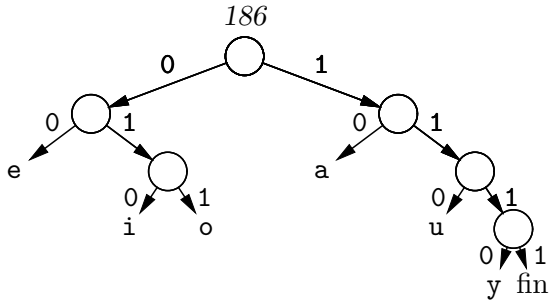
On choisit deux feuilles ayant la plus petite fréquence et on les regroupe en un arbre :



L'arbre construit reçoit comme fréquence la somme des fréquences des deux lettres qu'il regroupe. On répète ensuite ce procédé, regroupant à nouveau les deux arbres ayant la plus petite fréquence, jusqu'à obtenir finalement un arbre unique. Dans notre exemple, le regroupement des arbres de fréquences 11 et 20 donne :



On regroupe ensuite les arbres de fréquences 30 et 25, puis 31 et 40, puis 55 et 60. En regroupant les deux arbres restants, on obtient enfin l'arbre de Huffman recherché :



Traduisons maintenant cet algorithme en Caml.

Fichier `huffman.ml`

```
let construire_arbre fréquences =
  let prio = ref (fileprio__ajoute fileprio__vide 1 Fin) in
  let nombre_d'arbres = ref 1 in
  for c = 0 to 255 do
    if fréquences.(c) > 0 then begin
      prio := fileprio__ajoute !prio
                fréquences.(c) (Lettre(char_of_int c));
      incr nombre_d'arbres
    end
  done;
  for n = !nombre_d'arbres downto 2 do
    let (fréq1, arbre1, prio1) = fileprio__extraire !prio in
    let (fréq2, arbre2, prio2) = fileprio__extraire prio1 in
    prio := fileprio__ajoute prio2
              (fréq1 + fréq2) (Noeud(arbre1, arbre2))
  done;
  let (_, arbre, _) = fileprio__extraire !prio in
  arbre;;
```

Pour gérer l'ensemble d'arbres et les fréquences associées, on a utilisé les fonctions d'un module `fileprio` implémentant la structure de données connue sous le nom de file d'attente avec priorité. Voici l'interface de ce module :

Fichier `fileprio.mli`

```
type 'a t;;
value vide: 'a t
  and ajoute: 'a t -> int -> 'a -> 'a t
  and extraire: 'a t -> int * 'a * 'a t;;
exception File_vide;;
```

Les deux opérations de base sur une file d'attente avec priorité sont l'ajout d'un élément dans la file, avec une certaine priorité (un entier), et l'extraction d'un élément ayant la priorité la plus faible. L'opération d'extraction renvoie, en plus de l'élément extrait, la priorité de cet élément et la file d'attente privée de cet élément. Elle déclenche l'exception `File_vide` si on l'applique à la file vide. On donne en annexe de ce chapitre (section 13.4) deux implémentations possibles du module `fileprio`.

À partir de l'arbre de Huffman renvoyé par la fonction `construire_arbre`, on calcule une table de codage adaptée à la fonction `encode`, comme suit.

Fichier `huffman.ml`

```
let arbre_vers_codage arbre =
  let codage = { caractère = make_vect 256 []; fin = [] } in
  let rec remplir_codage préfixe = function
  | Lettre c ->
      codage.caractère.(int_of_char c) <- rev préfixe
  | Fin ->
      codage.fin <- rev préfixe
  | Noeud(arbre1, arbre2) ->
      remplir_codage (0 :: préfixe) arbre1;
      remplir_codage (1 :: préfixe) arbre2 in
  remplir_codage [] arbre;
  codage;;
```

La fonction locale `remplir_codage` effectue une exploration exhaustive de l'arbre. Son argument `préfixe` contient le chemin (inversé) de la racine jusqu'au nœud courant. Lorsqu'on atteint une feuille, on remet le chemin à l'endroit et on le stocke dans la case correspondante du codage.

Compression et décompression de fichiers

Il est temps de recoller tous les morceaux pour définir les fonctions de compression et de décompression d'un fichier. Le dernier problème à résoudre est celui du stockage du codage utilisé à la compression en tête du fichier compressé. Pour rester simple, nous stockons ce codage sous la forme de son arbre de Huffman, en utilisant les fonctions d'entrée-sortie structurée `output_value` et `input_value`. La fonction prédéfinie `output_value`, de type `out_channel -> 'a -> unit`, écrit une représentation de son argument sous forme d'une suite d'octets sur le canal spécifié. L'argument est une structure Caml quelconque (ou presque : elle ne doit pas contenir de valeurs fonctionnelles). La fonction `input_value` effectue l'opération inverse : elle lit une suite d'octets sur un canal et renvoie la structure Caml correspondante. Ces deux fonctions sont très rapides et évitent d'avoir à écrire soi-même des fonctions de conversion entre structures de données et suites d'octets. Le format de données utilisé pour stocker l'objet dans le fichier n'est pas du texte directement lisible par l'utilisateur, mais un codage binaire de l'objet. Il est destiné à être relu par des machines, non par des humains.

Un fichier compressé se compose donc d'un arbre de Huffman écrit par `output_value`, suivi par les données compressées écrites par `encode`.

Fichier `huffman.ml`

```
let compresse entrée sortie =
  let fréq = fréquences entrée in
  let arbre = construire_arbre fréq in
  let codage = arbre_vers_codage arbre in
  output_value sortie arbre;
  seek_in entrée 0;
  encode entrée sortie codage;;
```

La fonction prédéfinie `seek_in` positionne le pointeur de lecture d'un canal d'entrée. Le pointeur de lecture du canal `entrée`, qui est à la fin du fichier au retour de la fonction

`fréquences`, est donc remis au début du fichier par l'appel `seek_in entrée 0`. Cela permet à `encode` de relire les octets du fichier à compresser.

Fichier `huffman.ml`

```
let décompresse entrée sortie =
  let arbre = input_value entrée in
  décode entrée sortie arbre;;
```

13.4 Annexes

Pour finir le programme de compression, il reste à implémenter les files d'attente avec priorité (module `fileprio`) et les entrées-sorties bit à bit (module `esbit`).

Files d'attente avec priorité

On rappelle l'interface du module `fileprio` :

Fichier `fileprio.mli`

```
type 'a t;;
value vide: 'a t
  and ajoute: 'a t -> int -> 'a -> 'a t
  and extraire: 'a t -> int * 'a * 'a t;;
exception File_vide;;
```

Une première représentation des files d'attente consiste en des listes de paires (priorité, élément), triées par ordre de priorité croissante. De la sorte, l'élément de priorité minimale est toujours en tête de la liste et la fonction `extraire` est très simple.

Fichier `fileprio.ml`

```
type 'a t == (int * 'a) list;;
let vide = [];;
let extraire = function
  | [] -> raise File_vide
  | (prio, elt) :: reste -> (prio, elt, reste);;
```

La fonction `ajoute` est un peu plus compliquée : il faut parcourir la liste jusqu'à trouver un élément de priorité plus grande que l'élément à insérer.

Fichier `fileprio.ml`

```
let rec ajoute file prio elt =
  match file with
  | [] -> [(prio, elt)]
  | (prio1, elt1) :: reste ->
      if prio < prio1
      then (prio, elt) :: reste
      else (prio1, elt1) :: ajoute reste prio elt;;
```

Avec cette implémentation des files d'attente, l'opération `extraire` est en temps constant, mais l'opération `ajoute` est en temps $n/2$ en moyenne et n dans le pire des cas, où n est le nombre d'éléments de la file d'attente. Dans l'algorithme de Huffman, on

insère jusqu'à 256 éléments dans des files de 0 à 255 éléments, ce qui donne, au pire, à peu près 32000 tours dans la fonction `ajoute`. Il est à craindre que cette implémentation naïve ne soit pas assez efficace.

Voici une seconde implémentation du module `fileprio`, où les files sont représentées par des arbres tournoi (aussi appelés *heaps*, « tas » dans la littérature en anglais).

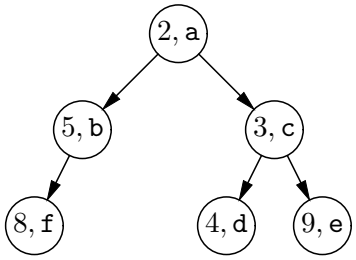
Fichier `fileprio.ml`

```
type 'a t = | Vide | File of int * 'a * 'a t * 'a t;;
let vide = Vide;;
```

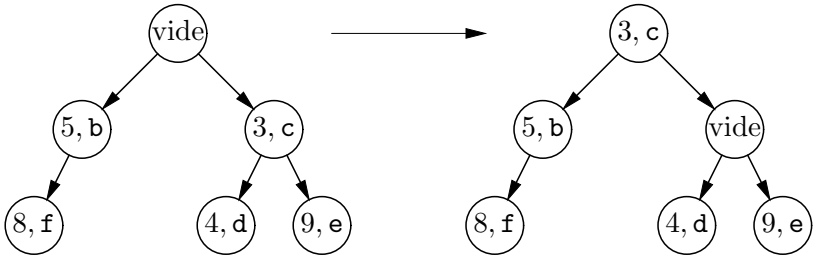
Les feuilles de l'arbre (constructeur `Vide`) ne portent pas d'information. Les nœuds de l'arbre (constructeur `File`) portent chacun un élément de la file, avec sa priorité, plus deux sous-arbres, traditionnellement appelés « le fils gauche » et « le fils droit ». On impose la condition suivante :

> La priorité d'un nœud est inférieure ou égale à la priorité de tous les nœuds contenus dans ses fils gauche et droit.

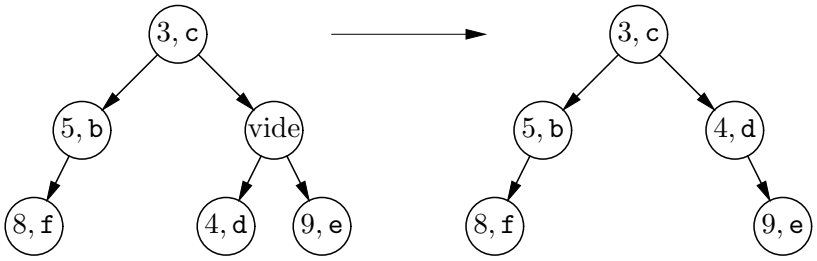
Ainsi, les priorités vont en croissant quand on se déplace de la racine vers une feuille. Voici un exemple d'arbre binaire croissant :



Comme dans le cas de l'implémentation par des listes ordonnées, l'élément de la file ayant la plus faible priorité est facile à trouver : c'est toujours l'élément à la racine de l'arbre. Pour implémenter `extraire`, il reste donc à savoir combiner les deux fils de la racine en un seul arbre binaire croissant, qui représente la file de départ privée de son élément le moins prioritaire. La racine du nouvel arbre est l'élément de plus petite priorité parmi ceux qui restent. Ce ne peut être que le sommet du fils gauche ou le sommet du fils droit, puisque tous les autres nœuds sont moins prioritaires. On déplace donc celui des deux sommets qui a la plus petite priorité, pour le mettre à la racine.



Bien entendu, il y a maintenant un trou dans l'un des deux fils, trou qu'il faut à son tour combler en répétant le même raisonnement.

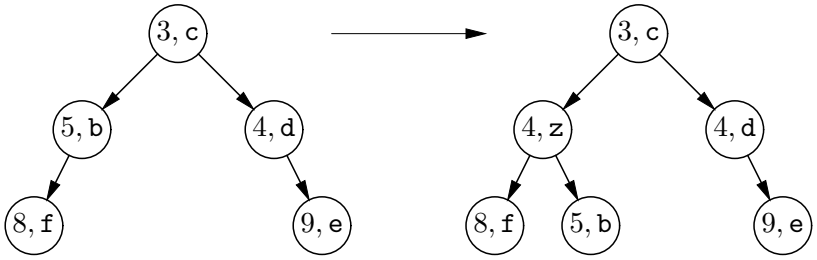


La procédure s'arrête lorsqu'elle atteint une feuille de l'arbre. Le résultat est bien un arbre binaire croissant contenant les mêmes éléments que celui de départ, moins la racine.

Fichier `fileprio.ml`

```
let rec enlève_sommet = function
  | Vide -> raise File_vide
  | File(prio, elt, Vide, Vide) -> Vide
  | File(prio, elt, gauche, Vide) -> gauche
  | File(prio, elt, Vide, droite) -> droite
  | File(prio, elt, (File(prio_g, elt_g, _, _) as gauche),
                    (File(prio_d, elt_d, _, _) as droite)) ->
      if prio_g < prio_d
      then File(prio_g, elt_g, enlève_sommet gauche, droite)
      else File(prio_d, elt_d, gauche, enlève_sommet droite);;

let extraire = function
  | Vide -> raise File_vide
  | File(prio, elt, _, _) as file -> (prio, elt, enlève_sommet file);;
```

L'ajout d'un élément à un arbre binaire croissant suit le même principe. Si le nouvel élément a une priorité plus haute que la racine, la fonction d'ajout s'appelle récursivement pour l'ajouter au fils gauche ou au fils droit. Si le nouvel élément est moins prioritaire que la racine, elle place le nouvel élément à la racine et s'appelle récursivement pour ajouter l'ancienne racine dans un des deux fils. La fonction s'arrête lorsqu'elle arrive sur une feuille. Voici un exemple d'insertion de `z` avec la priorité 4, où l'on choisit d'aller une fois à gauche, puis une fois à droite.



Dans la descente récursive, le choix entre fils gauche et fils droit est arbitraire. Cependant, il ne faut pas choisir toujours le fils gauche ou toujours le fils droit : après plusieurs ajouts, l'arbre obtenu serait très déséquilibré, avec une longue branche vers la gauche ou vers la droite. Pour que l'extraction et l'ajout soient efficaces, il faut que l'arbre ait les branches les plus courtes possible, c'est-à-dire qu'il soit aussi bien équilibré que

possible. Une manière d'obtenir ce résultat est d'insérer toujours du même côté (disons, à droite), mais de permuter les fils gauche et droit à chaque descente récursive. Cela suffit à garantir que l'arbre est toujours équilibré, même après une longue séquence d'ajouts.

Fichier `fileprio.ml`

```
let rec ajoute file prio elt =
  match file with
  | Vide ->
      File(prio, elt, Vide, Vide)
  | File(prio1, elt1, gauche, droite) ->
      if prio <= prio1
      then File(prio, elt, ajoute droite prio1 elt1, gauche)
      else File(prio1, elt1, ajoute droite prio elt, gauche);;
```

Pour ce qui est de l'efficacité de cette implémentation, on constate qu'un appel de `ajoute` ou de `extraire` parcourt au plus une branche entière de l'arbre. Comme l'arbre est équilibré, la longueur d'une branche est de l'ordre de $\log_2 n$, où n est le nombre d'éléments dans l'arbre. L'ajout et l'extraction se font donc en temps logarithmique en le nombre d'éléments de la file d'attente, ce qui donne une bien meilleure efficacité que l'implémentation à base de listes. En particulier, la construction de l'arbre de Huffman nécessite moins de 2000 tours dans la fonction `ajoute`, soit huit fois moins qu'avec l'implémentation naïve.

Entrées-sorties par bits

Nous passons maintenant aux entrées-sorties bit à bit (module `esbit`).

Fichier `esbit.mli`

```
value initialise : unit -> unit
  and écrire_bit : out_channel -> int -> unit
  and lire_bit : in_channel -> int
  and finir : out_channel -> unit;;
```

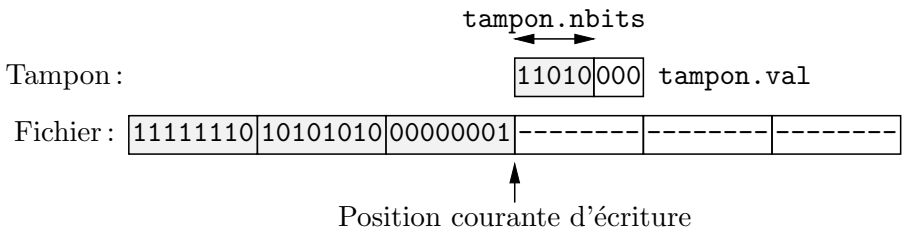
Les entrées-sorties standard de Caml Light présentent les fichiers comme une suite d'octets. Pour voir un fichier comme une suite de bits, nous allons traiter les octets comme des paquets de huit bits. Arbitrairement, on ordonne les bits des poids faibles vers les poids forts. Ainsi, l'entier 143, c'est-à-dire 10001111 en binaire, est vu comme la suite de huit bits $1, 1, 1, 1, 0, 0, 0, 1$.

Pour transformer des opérations bit à bit en opérations octet par octet, nous utilisons un tampon d'au plus huit bits contenant l'octet en cours d'écriture ou de lecture. Le tampon est représenté par un enregistrement à deux champs mutables, le champ `val` qui contient l'octet en cours et le champ `nbits` qui indique le nombre de bits valides dans cet octet.

Fichier `esbit.ml`

```
type tampon = { mutable val: int; mutable nbits: int };;
let tampon = { val = 0; nbits = 0 };;
let initialise () = tampon.val <- 0; tampon.nbits <- 0;;
```

Voici par exemple la situation en cours d'écriture. (On a marqué en grisé les bits qui ont été écrits par la fonction `écrire_bit`.)



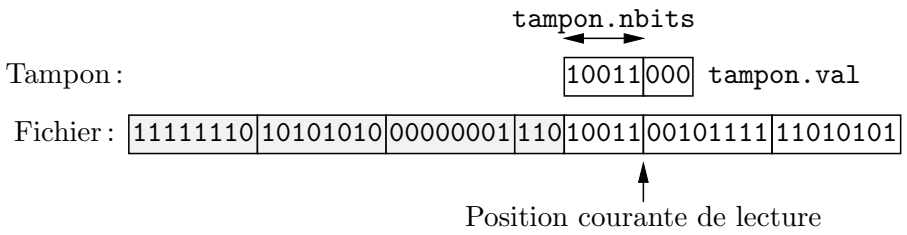
L'écriture d'un bit consiste simplement à le stocker dans le bit numéro `nbits` de `val`, puis à incrémenter `nbits`. Lorsque `nbits` atteint 8, on écrit l'octet `val` sur le fichier et on repart avec `nbits` valant 0.

Fichier `esbit.ml`

```
let écrire_bit sortie bit =
  tampon.val <- tampon.val lor (bit lsl tampon.nbits);
  tampon.nbits <- tampon.nbits + 1;
  if tampon.nbits >= 8 then begin
    output_char sortie (char_of_int tampon.val);
    tampon.val <- 0;
    tampon.nbits <- 0
  end;;

let finir sortie =
  if tampon.nbits > 0 then
    output_char sortie (char_of_int tampon.val);;
```

Les opérateurs infixes `lor` et `lsl` sont respectivement le « ou » bit à bit entre entiers et le décalage logique à gauche. En particulier, `bit lsl tampon.nbits` est un entier avec le bit numéro `tampon.nbits` égal à `bit` et tous les autres bits nuls.

La situation en cours de lecture est très symétrique. La seule différence est que `nbits` contient maintenant le nombre de bits restant à lire dans `val`. (On a marqué en grisé les bits qui ont été lus par la fonctions `lire_bit`.)



Fichier `esbit.ml`

```
let lire_bit entrée =
  if tampon.nbits <= 0 then begin
    tampon.val <- int_of_char(input_char entrée);
    tampon.nbits <- 8
  end;
  let res = tampon.val land 1 in
```

```
  tampon.val <- tampon.val lsr 1;
  tampon.nbits <- tampon.nbits - 1;
  res;;
```

Les opérateurs infixes `land` et `lsr` sont respectivement le « et » bit à bit entre entiers et le décalage logique à droite.

13.5 Mise en pratique

Tous les modules du programme sont maintenant écrits ; il reste à les compiler et à les lier entre eux.

```
$ camlc -c esbit.mli
$ camlc -c esbit.ml
$ camlc -c fileprio.mli
$ camlc -c fileprio.ml
$ camlc -c huffman.mli
$ camlc -c huffman.ml
$ camlc -c compr.ml
$ camlc -o compr esbit.zo fileprio.zo huffman.zo compr.zo
```

Pour essayer le programme :

```
$ camlrun compr monfichier
$ camlrun compr -d monfichier.cpr
```

Une autre manière de procéder est de compiler le programme à partir du système interactif, avec les commandes `compile` et `load_object`, comme décrit au chapitre 10, section 10.6. La compression s'effectue alors par `compr__compresse_fichier "fich"` et la décompression par `compr__décompresse_fichier "fich.cpr"`.

13.6 Pour aller plus loin

L'algorithme de Huffman employé ici n'est certainement pas le meilleur algorithme de compression existant à l'heure actuelle. Outre un taux de compression pas très élevé, il présente deux inconvénients pratiques : le fichier compressé doit contenir l'arbre de Huffman construit au codage et le fichier d'entrée doit être lu deux fois. Le premier trait est gênant pour les petits fichiers, pour lesquels la taille de l'arbre de Huffman n'est pas négligeable devant la taille du fichier produit ; en particulier, le fichier compressé peut devenir plus gros que le fichier d'entrée. Le deuxième trait implique que l'algorithme de Huffman n'est pas adapté à la compression "au vol" de données, comme dans le cas des données qui transitent à travers un modem et une ligne téléphonique.

Il existe une variante de l'algorithme de Huffman qui ne présente pas ces deux inconvénients : le codage de Huffman dynamique. L'idée est de changer d'arbre de Huffman en cours de compression. On part d'un arbre de Huffman équilibré, correspondant au cas où tous les caractères ont la même fréquence. On lit les caractères sur l'entrée, en tenant à jour les fréquences des caractères déjà lus. Chaque caractère est codé avec l'arbre de Huffman courant, puis l'arbre est modifié pour qu'il corresponde toujours aux fréquences des caractères déjà lus. Il n'est pas obligatoire de reconstruire

l'arbre à partir de zéro à chaque nouveau caractère : on arrive à modifier l'arbre de manière locale et incrémentale. Non seulement une deuxième passe sur l'entrée devient alors inutile, mais il n'est même plus nécessaire de transmettre l'arbre de Huffman au décompresseur : la décompression part du même arbre initial que la compression, et le modifie incrémentalement à chaque caractère décodé, en suivant exactement le même algorithme que le compresseur. De la sorte, l'arbre de décodage et l'arbre de codage évoluent en phase. Sur des fichiers assez longs, l'algorithme de Huffman dynamique atteint les mêmes taux de compression que l'algorithme de Huffman statique.

Pour atteindre des taux de compression plus élevés, il faut passer à une autre famille d'algorithmes de compression, dus à Lempel et Ziv, qui exploitent une autre source de redondance dans les fichiers de données : outre le fait que certains caractères apparaissent plus fréquemment que d'autres, on trouve aussi des séquences de plusieurs caractères qui apparaissent plus fréquemment que d'autres. Par exemple, dans un programme Caml, les mots-clés comme `let` ou `function` reviennent très souvent. L'idée des algorithmes de Lempel-Ziv est d'attribuer un code à ces chaînes plus fréquentes que les autres, et de les remplacer par leur code. Cette idée se prête à de nombreuses variantes, qui diffèrent par la méthode de reconnaissance des chaînes fréquentes et par la manière de les coder. Les compresseurs les plus efficaces combinent ces algorithmes avec l'algorithme de Huffman dynamique, réinjectant la sortie du premier dans l'entrée du second, pour tirer parti des deux types de redondance.

Bibliographie

Des livres entiers ont été consacrés aux divers algorithmes de compression ; voir par exemple *Data compression : methods and theory* de James Storer (Computer Science Press) et *Text compression* de Bell, Witten et Cleart (Prentice Hall). Pour ce qui est des algorithmes sur les files d'attente, on se reportera à l'un des classiques de l'algorithmique, comme par exemple le volume 3 de *The art of computer programming*, de Donald Knuth (Addison-Wesley).

14
Simulation d'un processeur

Où l'on apprend à gérer les RISC.

Beaucoup de programmes d'ordinateur servent à simuler des processus physiques, l'ordinateur offrant sa puissance de calcul pour faire évoluer les nombreux paramètres du système. Nous écrivons maintenant un programme de ce type : il simule le fonctionnement d'un processeur, ou unité centrale d'ordinateur. Le simulateur lit des programmes écrits dans le langage d'assemblage de notre processeur imaginaire et les exécute comme le ferait un processeur réel. Pour l'instant, nous écrirons les programmes assembleur à la main ; dans le chapitre 15, nous verrons comment produire automatiquement ces programmes par traduction à partir d'un langage de haut niveau.

Ce chapitre constitue une introduction au modèle de processeurs RISC (*reduced instruction set computer*, processeur à jeu d'instruction réduit). Le lecteur est supposé connaître dans ses grandes lignes le fonctionnement d'un processeur et avoir déjà programmé, ne serait-ce qu'une fois, dans un langage d'assemblage.

14.1 Le pico-processeur

Cette section décrit l'architecture et le langage du processeur que nous simulons. Nous l'appellerons le pico-processeur, car il est encore plus simple que la plupart des micro-processeurs actuels. Sous ses airs de simplicité, le pico-processeur reste néanmoins très proche de certains processeurs RISC bien connus, tels que l'Alpha ou le MIPS.

Architecture

Vu du programmeur, le pico-processeur se compose d'un ensemble de 32 registres, d'une mémoire de code dans laquelle est stocké le programme à exécuter et d'une mémoire de données dans laquelle le programme peut stocker des résultats intermédiaires.

Chaque registre contient un mot machine, que nous prenons de même taille que les entiers de Caml Light (type `int`), c'est-à-dire 31 bits. La plupart des machines actuelles

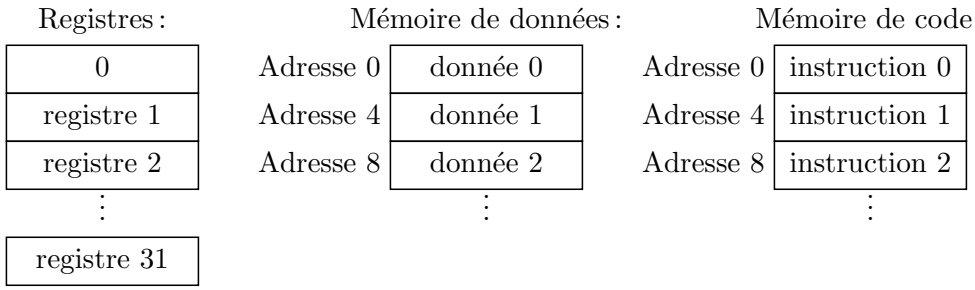


Figure 14.1: Le modèle du programmeur de la pico-machine

emploient des mots de 32 bits ; les mots de 31 bits sont irréalistes, mais simplifient considérablement la simulation.

Les registres sont numérotés de 0 à 31. Ils sont « interchangeables », en ceci que n'importe quel registre peut être argument ou résultat de n'importe quelle opération : il n'y a pas de registre spécifique à telle ou telle opération. Cependant, le registre 0 a une propriété particulière : il contient toujours l'entier zéro. Autrement dit, écrire dans ce registre n'a aucun effet : il garde sa valeur d'origine, c'est-à-dire 0. On peut douter de l'utilité d'un tel registre contenant toujours la même information. Et pourtant, il sert beaucoup : il permet de régulariser et de simplifier le jeu d'instructions, comme nous le verrons dans les exemples.

Tout comme le banc de registres, la mémoire de données est elle aussi organisée en mots. Comme sur les machines 32 bits modernes, les adresses des cases mémoire sont multiples de 4 : le premier mot est à l'adresse zéro, le deuxième à l'adresse 4, etc.

Enfin, la mémoire de code contient une instruction par case. Les adresses de code sont des entiers : très réalistement nos instructions sont toutes codées sur un mot machine ; leurs adresses sont donc aussi multiples de 4. Cependant, pour simplifier la simulation, nous les rangeons dans un tableau Caml et les représentons non pas par un entier mais par une valeur de type somme. Ainsi, la première instruction est à l'adresse zéro, mais rangée dans la case numéro 0 du tableau des instructions, la deuxième instruction est à l'adresse quatre, mais rangée dans la case numéro 1 du tableau des instructions, etc.

Cette division de la mémoire en deux zones n'est pas courante dans les processeurs. En général, la mémoire est constituée de mots ou d'octets où l'on range indifféremment des données ou des instructions. C'est encore une fois pour simplifier la simulation que nous avons divisé la mémoire en deux zones, pour faciliter le décodage des instructions, que nous gardons sous forme symbolique. L'alternative aurait été de coder effectivement les instructions sous la forme de nombres entiers et de les décoder au moment de leur exécution. La procédure de codage et de décodage n'est pas très intéressante et alourdirait inutilement notre présentation.

Jeu d'instructions

Le jeu d'instructions de la pico-machine est résumé dans la figure 14.2. Comme sur la plupart des processeurs RISC, les opérations arithmétiques et logiques ont un format

Notations : r nom de registre (`r 0`, `r 1`, ..., `r 31`)
o nom de registre ou constante entière (`12`, `-34`, ...)
a constante entière

Syntaxe	Instruction	Effet
`add`(r_1, o, r_2)	Addition entière	r_2 reçoit $r_1 + o$
`sub`(r_1, o, r_2)	Soustraction entière	r_2 reçoit $r_1 - o$
`mult`(r_1, o, r_2)	Multiplication entière	r_2 reçoit $r_1 * o$
`div`(r_1, o, r_2)	Quotient entier	r_2 reçoit r_1/o
`and`(r_1, o, r_2)	« Et » bit à bit	r_2 reçoit r_1 « et » o
`or`(r_1, o, r_2)	« Ou » bit à bit	r_2 reçoit r_1 « ou » o
`xor`(r_1, o, r_2)	« Ou exclusif » bit à bit	r_2 reçoit r_1 « ou exclusif » o
`shl`(r_1, o, r_2)	Décalage arithmétique logique à gauche	r_2 reçoit r_1 décalé à gauche de o bits
`shr`(r_1, o, r_2)	Décalage arithmétique logique à droite	r_2 reçoit r_1 décalé à droite de o bits
`slt`(r_1, o, r_2)	Test « inférieur »	r_2 reçoit 1 si $r_1 < o$, 0 sinon
`sle`(r_1, o, r_2)	Test « inférieur ou égal »	r_2 reçoit 1 si $r_1 \leq o$, 0 sinon
`seq`(r_1, o, r_2)	Test « égal »	r_2 reçoit 1 si $r_1 = o$, 0 sinon
`load`(r_1, o, r_2)	Lecture mémoire	r_2 reçoit le contenu de l'adresse $r_1 + o$
`store`(r_1, o, r_2)	Écriture mémoire	le contenu de r_2 est écrit à l'adresse $r_1 + o$
`jmp`(o, r)	Branchement	saute à l'adresse o et stocke l'adresse de l'instruction suivant le `jmp` dans r
`braz`(r, a)	Branchement si zéro	saute à l'adresse a si $r = 0$
`branz`(r, a)	Branchement si pas zéro	saute à l'adresse a si $r \neq 0$
`scall`(n)	Appel système	n est le numéro de l'appel
`stop`	Arrêt de la machine	fin du programme

Figure 14.2: Le jeu d'instructions de la pico-machine

simple et uniforme : le premier argument est un registre, le deuxième argument est soit un registre soit une constante et le résultat est mis dans un registre. Les opérations n'accèdent jamais directement à la mémoire.

Les transferts de données entre registres et mémoire de données sont assurés par les instructions `load` et `store`. L'adresse du mot mémoire accédé est toujours de la forme $r + o$, où r est le contenu d'un registre et o soit le contenu d'un registre, soit une constante. En d'autres termes, le pico-processeur n'a que deux modes d'adressage : indexé avec déplacement constant et indexé avec déplacement calculé.

Les instructions de branchement sont au nombre de trois. On a d'une part deux branchements conditionnels, `braz` et `branz`, qui testent si un registre est nul ou non nul et sautent, le cas échéant, à une adresse donnée. On dispose aussi d'un branchement inconditionnel, `jmp`, à une adresse constante ou calculée (contenue dans un registre). De plus, `jmp` écrit l'adresse de l'instruction qui le suit dans un registre. Ceci permet de

faire facilement des appels de sous-programmes (voir ci-dessous).

Enfin, nous avons ajouté deux instructions spéciales qui n'ont pas toujours d'équivalent dans les processeurs réels, mais qui sont très utiles dans le cadre d'un simulateur : `stop`, qui arrête l'exécution du programme, et `scall` (pour *system call*, appel système), qui représente l'appel à un (hypothétique) système d'exploitation, en particulier pour faire des entrées-sorties. Nous utiliserons deux appels système : l'un qui affiche à l'écran le nombre contenu dans le registre `r 1`, l'autre qui lit un nombre au clavier et le place dans `r 1`.

Idiomes de programmation

Le lecteur habitué aux architectures CISC (*complex instruction set computer*, processeur à jeu d'instructions complexe), comme par exemple le Pentium d'Intel, a sans doute été surpris par la pauvreté du jeu d'instructions du pico-processeur. En particulier, il semble manquer un certain nombre d'instructions indispensables : l'affectation (instruction `move`), certaines comparaisons (on a « inférieur », mais pas « supérieur »), l'appel de sous-programme, le retour de sous-programme, la gestion de la pile, ... Nous allons voir que toutes ces fonctionnalités s'expriment facilement en une ou deux instructions de la pico-machine.

Zéro comme argument Beaucoup d'opérations utiles s'obtiennent en fixant à zéro un des deux arguments d'une instruction, en prenant soit la constante 0, soit le registre `r 0` comme argument. Voici quelques exemples :

`add` $r_1, \texttt{0}, r_2$	Copie r_1 dans r_2 (instruction `move`)
`add r 0`$, n, r_2$	Met la constante n dans r_2 (instruction `move`)
`sub r 0`$, r_1, r_2$	Met l'opposé de r_1 dans r_2 (instruction `neg`)
`braz r 0`$, a$	Saute à l'adresse a
`load` $r_1, \texttt{0}, r_2$	Lit le mot à l'adresse (calculée) r_1
`load r 0`$, a, r_2$	Lit le mot à l'adresse (constante) a

Le registre zéro comme résultat Parfois, le résultat d'une opération est inutile. La manière standard de s'en débarrasser sans modifier aucun registre est de mettre `r 0` comme registre de destination. Par exemple, `jmp` a,`r 0` se branche à l'adresse a, sans mettre l'adresse de retour dans aucun registre.

Négation booléenne et inversion de tests En supposant les valeurs de vérité représentées par 0 pour « faux » et autre chose que 0 pour « vrai », l'instruction `seq` r_1,`r 0`$, r_2$ calcule la négation d'une valeur de vérité : si r_1 est « faux » (nul), r_2 est mis à « vrai » ; si r_1 est « vrai » (non nul), r_2 est mis à « faux ». Exemple d'application : le test « strictement plus grand » entre un registre r_1 et un registre ou une constante o, avec résultat dans r_2, se calcule par les deux instructions

`sle` r_1, o, r_2
`seq` r_2,`r 0`$, r_2$

La première instruction calcule la négation du résultat désiré (la négation de $r_1 > o$ est $r_1 \leq o$) ; la deuxième calcule la négation de cette négation, obtenant le résultat désiré.

Sous-programmes L'instruction `jmp` sert à la fois pour appeler un sous-programme et pour revenir d'un sous-programme dans le programme principal. L'idée est de passer au sous-programme son adresse de retour dans un registre particulier. Par convention, nous utiliserons toujours le registre `r 31` pour passer l'adresse de retour et les registres `r 1`, `r 2`, ... pour passer les arguments et les résultats entre un sous-programme et le programme qui l'appelle. (Ce n'est qu'une convention, dans le but de rendre les programmes plus lisibles ; rien dans l'architecture de la pico-machine n'impose ce choix de registres.) Avec cette convention, voici comment s'écrit la fonction « moyenne arithmétique » :

```
Instruction 100  add r 1, r 2, r 1
            104  div r 1, 2, r 1
            108  jmp r 31, r 0
```

Et voici un programme d'essai qui appelle cette fonction :

```
Instruction 0    scall 0 (lecture d'un nombre au clavier)
            4    add r 1, 0, r 2
            8    scall 0 (lecture d'un nombre au clavier)
            12   jmp 100, r 31
            16   scall 1 (écriture d'un nombre à l'écran)
            20   stop
```

L'appel système `scall 0` lit un nombre au clavier et le place dans `r 1`. Les instructions 0, 4 et 8 lisent donc deux nombres et les placent dans les registres `r 2` et `r 1`. L'instruction 12 se branche à l'instruction 100 (le début du sous-programme), après avoir placé l'adresse de l'instruction 16 dans le registre `r 31`. On exécute alors le code de la fonction « moyenne arithmétique » : les instructions 100 et 104 calculent la moyenne de `r 1` et de `r 2` et placent le résultat dans `r 1`, conformément aux conventions d'appel des fonctions ; l'instruction 108 se branche à l'adresse contenue dans `r 31`, c'est-à-dire à l'instruction 16. On continue donc l'exécution du programme principal : affichage du registre `r 1` (le résultat) par l'appel système `scall 1` et arrêt du programme.

Gestion d'une pile Les conventions d'appel introduites ci-dessus posent problème lorsqu'un sous-programme doit en appeler un autre, ou se rappeler lui-même : si un sous-programme f appelle un sous-programme g avec les conventions standard, f va mettre dans `r 31` une adresse de retour pour g, détruisant ainsi le contenu courant de `r 31`, qui est l'adresse de retour pour f. La solution est bien sûr de sauvegarder l'adresse de retour pour f sur une pile.

La pico-machine ne fournit pas d'instructions `push` et `pop` pour gérer une pile ; mais ces deux opérations se programment facilement. On convient d'utiliser le registre `r 30` comme pointeur de pile. La pile commence tout en haut de la mémoire et croît vers le bas. Le registre `r 30` pointe vers le dernier mot empilé. Avec ces conventions, empiler un registre, disons `r 31`, se traduit par

```
sub r 30, 4, r 30
store r 30, 0, r 31
```

L'instruction `sub` alloue de la place pour le registre à empiler ; l'instruction `store` stocke sa valeur à l'emplacement alloué. Réciproquement, dépiler `r 31` se traduit par

```
load r 30, 0, r 31
add r 30, 4, r 30
```

L'instruction `load` recharge la valeur du registre ; l'instruction `add` récupère la place qui lui était allouée sur la pile.

Comme exemple d'utilisation de la pile, voici un sous-programme qui calcule la fonction factorielle de la manière récursive classique.

Adr.	Instruction	Commentaire
100	`branz r 1, 112`	si l'argument n n'est pas nul, aller en 112
104	`add r 0, 1, r 1`	mettre la constante `1` dans `r 1`
108	`jmp r 31, r 0`	retour à l'appelant
112	`sub r 30, 8, r 30`	réserver deux mots dans la pile
116	`store r 30, 4, r 31`	empiler `r 31` (l'adresse de retour)
120	`store r 30, 0, r 1`	et empiler `r 1` (n)
124	`sub r 1, 1, r 1`	appel récursif sur $n-1$
128	`jump 100, r 31`	au retour, `r 1` contient $(n-1)!$
132	`load r 30, 0, r 2`	dépile n (mis dans `r 2`)
136	`load r 30, 4, r 31`	dépile l'adresse de retour
140	`add r 30, 8, r 30`	récupère la place en pile
144	`mult r 1, r 2, r 1`	calcule $n \times (n-1)!$ dans `r 1`
148	`jmp r 31, r 0`	retour à l'appelant

14.2 Le simulateur

Nous passons maintenant à l'implémentation en Caml d'un simulateur de la pico-machine. Cette implémentation se compose de deux programmes : l'un, le simulateur proprement dit, prend une représentation binaire du code à exécuter et l'exécute ; l'autre, appelé l'assembleur, produit la représentation binaire exécutable à partir d'un texte de programme écrit dans le langage de la pico-machine. Nous étudierons l'assembleur dans la section 14.3 ; pour l'instant, voyons le simulateur.

Description du pico-processeur

On commence par un module `code` qui définit le jeu d'instructions de la machine sous forme d'un type concret Caml, ainsi que certaines caractéristiques de la machine.

Fichier `code.mli`

```
type registre == int;;

type opérande =
   | Reg of registre
   | Imm of int;;

type instruction =
   | Op of opération * registre * opérande * registre
   | Jmp of opérande * registre
   | Braz of registre * int
   | Branz of registre * int
```

```
   | Scall of int
   | Stop

and opération =
  | Load | Store | Add | Mult | Sub | Div
  | And | Or | Xor | Shl | Shr
  | Slt | Sle | Seq;;

value nombre_de_registres: int
  and sp: int
  and ra: int
  and taille_du_mot: int;;
```

Les instructions sont décrites par un ensemble de types somme et une abréviation de type (`type registre == int`). Une abréviation de type n'a pas de contenu sémantique : le type figurant à gauche du symbole `==` peut être partout remplacé par le type qui figure à droite. Dans notre cas, le type `registre` peut être partout remplacé par `int`. On utilise des abréviations de type dans le but de rendre le code plus clair.

La simplicité du jeu d'instructions de la machine se reflète dans la simplicité du type `opération` : il n'y a que 14 opérations de base. Dans les instructions, ces opérations sont regroupées dans le constructeur `Op`, puisque leurs arguments ont exactement la même forme (un registre source, un opérande source, un registre destination). Il y a cinq formats d'instructions puisque `Braz` et `Branz` ont les mêmes arguments.

L'implémentation du module `code` fixe le nombre de registres à 32 et donne des noms aux registres 30 (`sp`, pour *stack pointer*, pointeur de pile) et 31 (`ra`, pour *return address*, adresse de retour).

Fichier `code.ml`

```
let nombre_de_registres = 32
and sp = 30
and ra = 31
and taille_du_mot = 4;;
```

L'état du pico-processeur

Le module `simul` implémente le simulateur du pico-processeur : il fournit la fonction `exécute` qui exécute tout un programme, c'est-à-dire un tableau d'instructions, en initialisant d'abord la mémoire à une taille donnée.

Fichier `simul.mli`

```
#open "code";;

exception Erreur of string * int;;

value lire_mémoire : int -> int;;
value écrire_mémoire : int -> int -> unit;;
value lire_registre : int -> int;;
value écrire_registre : int -> int -> unit;;
value tableau_des_appels_système: (int -> int) vect;;
```

```
value exécute: instruction vect -> int -> unit;;
```

L'implémentation de `simul` commence par une description Caml de l'état courant de la pico-machine. Cet état se compose de l'état des registres, de la mémoire de code, de la mémoire de données, et enfin de la valeur courante du pointeur de programme (`pc`, pour *program counter*), qui contient l'adresse de la prochaine instruction à exécuter. Cet état est donc décrit par un type enregistrement à champs mutables, contenant un tableau pour les registres, le pointeur de code, un tableau pour les instructions à exécuter et un tableau pour la mémoire des données.

Fichier `simul.ml`

```
#open "code";;

type état_du_processeur =
   { registres: int vect;
     mutable pc: int;
     mutable code: instruction vect;
     mutable mémoire: int vect };;

let pico =
  { registres = make_vect nombre_de_registres 0;
    pc = 0;
    code = [| |];
    mémoire = [| |] };;
```

La pico-machine est créée par la définition de la variable `pico`. Ses zones code et mémoire des données sont provisoirement initialisées vides (elles ne seront allouées qu'au lancement de la machine, en fonction de la taille du programme à exécuter et de la taille des données à traiter).

Exécution d'une instruction

Le simulateur fait évoluer l'état du processeur, contenu dans la variable `pico`, en simulant l'exécution des instructions. Pour cela, le simulateur appelle la fonction `cycle_d'horloge` qui exécute une instruction et fait évoluer l'état de la machine en conséquence. L'exécution d'un programme consiste à répéter `cycle_d'horloge` jusqu'à ce qu'on rencontre une instruction `stop`.

La fonction `cycle_d'horloge` devra lire et écrire la mémoire et les registres. Nous définissons tout d'abord une batterie de fonctions auxiliaires qui vérifient que les opérations demandées par le programme sont légales et accèdent à des ressources réelles de la machine.

Fichier `simul.ml`

```
let lire_registre reg =
    if reg < 0 || reg >= nombre_de_registres then
      raise (Erreur ("registre illégal", reg));
    pico.registres.(reg);;

let écrire_registre reg valeur =
    if reg < 0 || reg >= nombre_de_registres then
```

```
      raise (Erreur ("registre illégal", reg));
    if reg <> 0 then pico.registres.(reg) <- valeur;;

let lire_instruction adresse =
    let adr = adresse / taille_du_mot in
    if adr < 0 || adr >= vect_length pico.code then
      raise (Erreur ("sortie de la zone code", adr));
    if adresse mod taille_du_mot <> 0 then
      raise (Erreur ("pc non aligné", adresse));
    pico.code.(adr);;

let lire_mémoire adresse =
    let adr = adresse / taille_du_mot in
    if adr < 0 || adr >= vect_length pico.mémoire then
      raise (Erreur ("lecture en dehors de la mémoire", adresse));
    if adresse mod taille_du_mot <> 0 then
      raise (Erreur ("lecture non alignée", adresse));
    pico.mémoire.(adr);;

let écrire_mémoire adresse valeur =
    let adr = adresse / taille_du_mot in
    if adr < 0 || adr >= vect_length pico.mémoire then
      raise (Erreur ("écriture en dehors de la mémoire", adresse));
    if adresse mod taille_du_mot <> 0 then
      raise (Erreur ("écriture non alignée", adresse));
    pico.mémoire.(adr) <- valeur;;

let valeur_opérande = function
  | Reg r -> lire_registre r
  | Imm n -> n;;
```

Dans le cas des accès à la mémoire, il faut tenir compte du fait que l'adresse d'un mot est toujours multiple de 4 ; il faut donc la diviser par 4 pour obtenir l'indice qui lui correspond dans le tableau `pico.mémoire`.

Fichier `simul.ml`

```
let tableau_des_appels_système =
  make_vect 10 ((function x -> x) : int -> int);;

let exécute_appel_système appel argument =
    if appel < 0 || appel >= vect_length tableau_des_appels_système
     then raise(Erreur("mauvais appel système", appel))
     else tableau_des_appels_système.(appel) argument;;
```

La fonction `cycle_d'horloge` exécute une instruction à la fois : elle décode et exécute l'instruction actuellement pointée par le compteur ordinal (PC), puis incrémente ce compteur pour passer à l'instruction suivante. Dans cette fonction, les opérations de la machine sont simulées à l'aide de fonctions Caml : `+`, `-`, `*`, et ainsi de suite pour les opérations arithmétiques et logiques ; `lire_mémoire` et `écrire_mémoire` pour `Load` et `Store`. L'arrêt du processeur, lorsqu'on rencontre l'instruction `Stop`, est modélisé par un déclenchement d'exception (l'exception `Arrêt`).

Fichier `simul.ml`

```
exception Arrêt;;

let cycle_d'horloge () =
  let instruction = lire_instruction pico.pc in
  pico.pc <- pico.pc + taille_du_mot;
  match instruction with
  | Op(opération, reg1, opérande, reg2) ->
      let arg1 = lire_registre reg1
      and arg2 = valeur_opérande opérande in
      begin match opération with
      | Load  -> écrire_registre reg2 (lire_mémoire (arg1 + arg2))
      | Store -> écrire_mémoire (arg1 + arg2) (lire_registre reg2)
      | Add   -> écrire_registre reg2 (arg1 + arg2)
      | Mult  -> écrire_registre reg2 (arg1 * arg2)
      | Sub   -> écrire_registre reg2 (arg1 - arg2)
      | Div   -> if arg2 = 0
                 then raise (Erreur("division par zéro", pico.pc-1))
                 else écrire_registre reg2 (arg1 / arg2)
      | And   -> écrire_registre reg2 (arg1 land arg2)
      | Or    -> écrire_registre reg2 (arg1 lor arg2)
      | Xor   -> écrire_registre reg2 (arg1 lxor arg2)
      | Shl   -> écrire_registre reg2 (arg1 lsl arg2)
      | Shr   -> écrire_registre reg2 (arg1 asr arg2)
      | Slt   -> écrire_registre reg2 (if arg1 < arg2 then 1 else 0)
      | Sle   -> écrire_registre reg2 (if arg1 <= arg2 then 1 else 0)
      | Seq   -> écrire_registre reg2 (if arg1 = arg2 then 1 else 0)
      end
  | Jmp(opérande, reg) ->
      écrire_registre reg pico.pc;
      pico.pc <- valeur_opérande opérande
  | Braz(reg, adresse) ->
      if lire_registre reg = 0 then pico.pc <- adresse
  | Branz(reg, adresse) ->
      if lire_registre reg <> 0 then pico.pc <- adresse
  | Scall(appel_système) ->
      écrire_registre 1
        (exécute_appel_système appel_système (lire_registre 1))
  | Stop -> raise Arrêt;;
```

Exécution d'un programme complet

L'exécution d'un programme complet (fonction `exécute`) consiste à charger la mémoire avec le programme, à allouer la mémoire des données, à initialiser le pointeur de pile et le compteur ordinal, puis à lancer une boucle sans fin de cycles d'horloge, qui ne s'arrête qu'à la rencontre de l'exception `Arrêt`. Enfin, puisque nous avons choisi par convention que le pointeur de pile serait le registre 30 et que la pile croîtrait vers le bas de la mémoire, le registre `sp` est initialisé à la plus grande adresse mémoire possible (plus un) au démarrage de la machine.

Fichier `simul.ml`

```
let exécute programme taille_mémoire_en_octets =
    let taille_mémoire_en_mots = (taille_mémoire_en_octets + 3) / 4 in
    pico.code <- programme;
    pico.mémoire <- make_vect taille_mémoire_en_mots 0;
    pico.registres.(0) <- 0;
    pico.registres.(sp) <- taille_mémoire_en_mots * taille_du_mot;
    pico.pc <- 0;
    try while true do cycle_d'horloge () done
    with Arrêt -> ();;
```

Les appels système

Il nous reste à mettre en place un « pico-système d'exploitation », à savoir les appels système pour la lecture et l'écriture d'un nombre. Nous définissons donc les deux fonctions correspondantes et les rangeons dans le tableau des appels système.

Fichier `simul.ml`

```
let appel_système_read _ =
    try read_int ()
    with Failure _ -> raise (Erreur ("erreur de lecture", 1))

and appel_système_write argument =
    print_int argument; print_newline (); argument;;

tableau_des_appels_système.(0) <- appel_système_read;
tableau_des_appels_système.(1) <- appel_système_write;;
```

Dans un vrai processeur les appels système sont bien sûr écrits en assembleur et accèdent directement aux ressources matérielles de la machine. Par exemple, la primitive d'impression irait (plus ou moins directement) écrire dans la mémoire d'écran de la machine chacun des chiffres du nombre, tandis que la routine de lecture d'un nombre interrogerait le clavier et transformerait les codes des touches frappées en un nombre entier à l'aide d'une boucle similaire à celles utilisées dans nos analyseurs syntaxiques. Pour simplifier, c'est Caml qui nous permet cet accès direct aux ressources de la machine, ce qui nous évite de décrire la connexion du processeur avec ses périphériques.

Le programme principal

Pour lancer la machine sur un fichier de code préalablement assemblé, on se contente de lire ce code en mémoire, puis de lancer la fonction `exécute`. Pour simplifier la lecture du code machine, nous allons utiliser les fonctions d'entrée-sortie structurée que fournit le système Caml Light. La fonction prédéfinie `output_value`, qui a pour type `out_channel -> 'a -> unit`, écrit une représentation de son argument (une valeur Caml quelconque) sous forme d'une suite d'octets sur le canal spécifié. La fonction `input_value` effectue l'opération inverse : elle lit une suite d'octets sur un canal et renvoie la structure Caml correspondante. (Ces deux fonctions nous ont déjà servi pour sauvegarder des arbres de Huffman à la section 13.3.)

Le point d'entrée de la commande analyse les arguments fournis par l'utilisateur pour détecter la présence d'un entier fixant la taille mémoire à utiliser ; sinon on lance la machine avec une taille de 1K mots. On surveille aussi les exceptions qui pourraient se produire pour afficher un message et rendre un code d'erreur adéquat.

Fichier `exec.ml`

```
#open "code";;
#open "simul";;

exception Fichier_incorrect;;

let exécute_fichier nom_fichier taille_mémoire =
    let canal = open_in_bin nom_fichier in
    let programme =
      try (input_value canal : instruction vect)
      with Failure _ -> raise Fichier_incorrect in
    close_in canal;
    exécute programme taille_mémoire;;

exception Mauvais_arguments;;

if sys__interactive then () else
try
  if vect_length sys__command_line < 2 then raise Mauvais_arguments;
  let taille_mémoire =
    if vect_length sys__command_line < 3
    then 1024
    else try int_of_string sys__command_line.(2)
         with Failure _ -> raise Mauvais_arguments in
  exécute_fichier sys__command_line.(1)
                  (taille_du_mot * taille_mémoire);
  exit 0
with Mauvais_arguments ->
       prerr_endline "Usage: pico_run <fichier> [taille mémoire]";
       exit 2
   | Fichier_incorrect ->
       prerr_endline "Le fichier ne contient pas du code exécutable";
       exit 2
   | Erreur(message, param) ->
       prerr_string "Erreur à l'exécution: ";
       prerr_string message;
       prerr_string " ("; prerr_int param; prerr_endline ")";
       exit 2
   | sys__Sys_error message ->
       prerr_string "Erreur du système: "; prerr_endline message;
       exit 2;;
```

Le programme `pico_run` s'obtient par une série d'appels au compilateur indépendant, suivie d'un appel à l'éditeur de liens de Caml Light qui produit la commande elle-même.

```
$ camlc -c code.mli
```

```
$ camlc -c code.ml
$ camlc -c simul.mli
$ camlc -c simul.ml
$ camlc -c exec.ml
$ camlc -o pico_run code.zo simul.zo exec.zo
```

14.3 L'assembleur

Nous passons maintenant au programme qui se charge de transformer un texte écrit dans le langage du processeur en une suite d'instructions directement compréhensibles par le processeur. Ce programme est traditionnellement appelé l'assembleur. Il produit une suite d'instructions directement exécutables, qu'il écrit dans un fichier de résultat. Ce fichier est ensuite lu et chargé en mémoire programme à la demande de l'utilisateur, par un programme spécial qui lance l'exécution. En général ce « lanceur » est le système d'exploitation de l'ordinateur ; dans notre cas, c'est le programme `pico_run` de la section précédente.

Le langage de l'assembleur

Le langage d'entrée de l'assembleur s'appelle en termes précis le langage d'assemblage et par abus l'assembleur. Le mot « assembleur » a donc deux sens en informatique : il désigne tantôt un langage, tantôt un programme de traduction. Pour éviter toute confusion, nous emploierons « langage d'assemblage » pour le langage d'instructions symboliques et « assembleur » pour le programme.

Le principal travail de l'assembleur est de lire une représentation textuelle des instructions de la machine et de la transformer en code exécutable. Dans un processeur réel, le code exécutable est une suite d'octets ou de mots qui encodent les instructions et leurs opérandes sous forme binaire. Pour la pico-machine, la phase de transformation du programme source en suite de nombres sera remplacée par la traduction en syntaxe abstraite des instructions de la pico-machine. Par exemple on écrira `store sp, 1, r 1` dans le langage d'assemblage et l'assembleur de la pico-machine produira l'instruction `Op (Store, 30, Imm 1, 1)`. Comme pour un processeur réel, l'assembleur effectue un codage des opérations ; au contraire d'un processeur réel, le code n'est pas sous forme binaire. Il aurait été possible de coder réellement en nombres entiers, au prix d'inutiles complications.

Un autre service que rend l'assembleur est de libérer le programmeur de la gestion des numéros d'instructions. On écrit les instructions à la suite et l'assembleur les range automatiquement par numéros croissants. Bien plus, il fournit la possibilité de repérer des instructions par des noms ; ce sont les étiquettes symboliques, qui font référence à des adresses d'instructions, sans que le programmeur ait à calculer l'adresse absolue de l'instruction correspondante dans la zone code. Ce remplacement automatique d'adresses symboliques par des adresses absolues s'appelle la résolution des étiquettes.

Le langage d'assemblage, tel qu'il est défini par l'assembleur, est donc plus riche et plus expressif que le langage exécuté par la machine, puisqu'il comporte ces étiquettes symboliques. L'assembleur fournit également un certain nombre d'abréviations ; par exemple, `sp` est un nom de registre légal de l'assembleur, qu'il transforme automatique-

ment en une référence au registre numéro 30. Il s'agit là encore de noms symboliques, automatiquement gérés par l'assembleur. Enfin, il enrichit et rend plus uniforme le jeu d'instructions de la machine, en ajoutant des pseudo-instructions qui se présentent exactement comme des instructions de la machine mais sont en fait expansées en une ou plusieurs « vraies » instructions. Par exemple, notre assembleur sait coder les comparaisons « supérieur » et « supérieur ou égal » (instructions `sgt` et `sge`), qui sont initialement absentes du jeu d'instructions.

Voici par exemple un fichier d'assembleur de la pico-machine, écrit à la main et qui programme la fonction factorielle.

Fichier `fact.asm`

```
# Le programme principal
        read                    # lecture de l'argument (dans r 1)
        jmp     fact, ra        # calcul de la factorielle
        write                   # écriture du résultat (r 1)
        stop

# La fonction factorielle(N)
# L'argument N est dans r 1. Le résultat est mis dans r 1.
fact:   braz    r 1, fact_0     # N = 0 ?
        sub     sp, 8, sp       # réserve deux places dans la pile
        store   sp, 0, ra       # sauvegarde de l'adresse de retour
        store   sp, 4, r 1      # et de la valeur de N
        sub     r 1, 1, r 1
        jmp     fact, ra        # appel récursif sur N-1
        load    sp, 4, r 2      # récupération de la valeur de N
        mult    r 1, r 2, r 1   # calcul de N * fact(N-1)
        load    sp, 0, ra       # récupération de l'adresse de retour
        add     sp, 8, sp       # et de la place en pile
        jmp     ra, r 0         # retour à l'appelant
fact_0: add     r 0, 1, r 1     # mettre 1 dans r1
        jmp     ra, r 0         # retour à l'appelant
```

On a écrit simplement `jmp fact, ra` en utilisant l'étiquette symbolique définie par `fact:` au lieu de `jmp 16, r 31` qui ferait référence au numéro absolu de l'instruction `braz r 1, fact_0` et au numéro du registre dédié à l'adresse de retour.

En observant le code assembleur de la fonction factorielle, on mesure la difficulté qu'il y a à écrire un programme en assembleur plutôt qu'en Caml. On comprend aussi bien mieux la différence entre style impératif et style fonctionnel : l'assembleur est par essence impératif, car on ne travaille que par modification de registres. Vous constatez aussi qu'on doit tout gérer soi-même, « à la main », par exemple la récursivité dans le cas de `fact`. En revanche, en assembleur toutes les ressources de la machine sont disponibles : on peut tout faire ; malheureusement, la contrepartie est qu'il faut tout faire soi-même.

Stockage du code et gestion des étiquettes

Fichier `stockage.mli`

```
#open "code";;

exception Erreur of string;;

value initialise: unit -> unit
  and assemble: instruction -> unit
  and poser_étiquette: string -> unit
  and valeur_étiquette: string -> int
  and extraire_code: unit -> instruction vect;;
```

Pour résoudre les étiquettes, l'assembleur est obligé de fonctionner en deux étapes : dans la première étape il « pose » les étiquettes quand il les rencontre, c'est-à-dire qu'il note leur adresse absolue dans une table d'association. Lorsqu'il rencontre à nouveau l'étiquette il la remplace par sa valeur. Malheureusement, cette phase ne suffit pas, car le programmeur peut faire référence à des étiquettes « en avant », c'est-à-dire encore inconnues car pas encore rencontrées. C'est le cas de l'instruction `jmp fact, r 31`, qui est assemblée alors que l'étiquette `fact` sera lue trois instructions plus loin. Dans ce cas, l'assembleur laisse l'adresse de l'étiquette à zéro et note que l'instruction assemblée est à compléter lorsque l'étiquette sera connue. C'est le rôle de la seconde phase d'assemblage que de repasser sur ces instructions incomplètes et de les modifier avec les adresses désormais déterminées. La fonction `résoudre_étiquette` se charge de ce travail (appelé *backpatching* en anglais).

Pour obtenir l'adresse absolue des étiquettes, l'assembleur gère un compteur ordinal virtuel, qu'il incrémente à chaque nouvelle instruction assemblée. Il engrange ses résultats dans un tableau d'instructions assemblées. Ces quantités font partie de l'état de l'assembleur, avec la table des étiquettes et la liste des étiquettes à résoudre. La table des étiquettes est une table de hachage comme celles décrite à la section 12.8.

Fichier `stockage.ml`

```
#open "code";;

type état_de_l'assembleur =
   { mutable pc: int;
     mutable code: instruction vect;
     table_étiq: (string, int) hashtbl__t;
     mutable à_résoudre: (int * string) list };;

let asm =
  { pc = 0; code = [||]; table_étiq = hashtbl__new 17;
    à_résoudre = [] };;

let initialise () =
    asm.pc <- 0;
    asm.code <- make_vect 100 Stop;
    hashtbl__clear asm.table_étiq;
    asm.à_résoudre <- [];;

let décode_adresse adr = adr / taille_du_mot;;
```

```
let assemble instruction =
    if asm.pc >= vect_length asm.code then begin
      let nouveau_code = make_vect (2 * vect_length asm.code) Stop in
      blit_vect asm.code 0 nouveau_code 0 (vect_length asm.code);
      asm.code <- nouveau_code
    end;
    asm.code.(décode_adresse asm.pc) <- instruction;
    asm.pc <- asm.pc + taille_du_mot;;

let définir_étiquette nom_étiq val_étiq =
    try
      hashtbl__find asm.table_étiq nom_étiq;
      raise (Erreur ("étiquette " ^ nom_étiq ^ " redéfinie"))
    with Not_found ->
      hashtbl__add asm.table_étiq nom_étiq val_étiq;;

let poser_étiquette nom_étiq =
    définir_étiquette nom_étiq asm.pc;;

let valeur_étiquette nom_étiq =
    try
       hashtbl__find asm.table_étiq nom_étiq
    with Not_found ->
       asm.à_résoudre <- (asm.pc, nom_étiq) :: asm.à_résoudre;
       0;;
```

La fonction `assemble` surveille le compteur ordinal virtuel : s'il déborde de la mémoire programme virtuelle alors on remplace le tableau initial par un nouveau tableau deux fois plus long, dans lequel on recopie les instructions déjà assemblées, et l'on continue normalement. (La recopie est effectuée par la fonction prédéfinie `blit_vect`, qui est l'analogue pour les tableaux de la fonction `blit_string` pour les chaînes.) Il s'agit là d'une extension de la taille de la mémoire virtuelle de l'assembleur : lorsque le code est complètement assemblé, l'assembleur pourra déterminer exactement la taille du programme, qui sera celle qu'on attribuera à l'exécution. En effet, dans le pico-processeur (comme dans le monde réel), la mémoire programme est fixée une fois pour toutes au lancement. Il n'est pas possible de changer la taille de la zone programme pendant que la pico-machine est en marche. D'ailleurs pourquoi en aurait-on besoin, puisqu'il s'agit alors d'exécuter un certain programme fixé.

Nous détaillons maintenant le code de la fonction qui résout les étiquettes en modifiant les instructions où elles sont apparues alors qu'on ne connaissait pas encore leur valeur. Les étiquettes qui repèrent une instruction dans la mémoire programme peuvent apparaître dans les instructions de branchement, donc comme argument des instructions `Jmp`, `Braz` ou `Branz`. Ce sont les trois derniers cas du filtrage qui définit la variable `nouvelle_instruction`, qui a évidemment pour valeur l'instruction provisoirement écrite par l'assembleur, mais avec la valeur maintenant connue de l'étiquette. Les étiquettes peuvent aussi apparaître dans des opérations, comme deuxième argument constant : en effet, il est parfois nécessaire de lire ou d'écrire en mémoire des données l'adresse d'une instruction (par exemple pour écrire directement l'adresse de

retour d'une fonction sur la pile).

Fichier `stockage.ml`

```
let résoudre_étiquette (adresse, nom_étiq) =
    let valeur =
        try
          hashtbl__find asm.table_étiq nom_étiq
        with Not_found ->
          raise (Erreur ("étiquette " ^ nom_étiq ^ " indéfinie")) in
    let nouvelle_instruction =
        match asm.code.(décode_adresse adresse) with
        | Op(opération, reg1, _, reg2) ->
            Op(opération, reg1, Imm valeur, reg2)
        | Jmp(_, reg) ->
            Jmp(Imm valeur, reg)
        | Braz(reg, _) ->
            Braz(reg, valeur)
        | Branz(reg, _) ->
            Branz(reg, valeur)
        | _ -> raise (Erreur "résoudre_étiquette") in
    asm.code.(décode_adresse adresse) <- nouvelle_instruction;;

let extraire_code () =
    do_list résoudre_étiquette asm.à_résoudre;
    sub_vect asm.code 0 (décode_adresse asm.pc);;
```

Finalement, la fonction `extraire_code` appelle `résoudre_étiquette` sur la liste des étiquettes non résolues, puis renvoie le tableau des instructions assemblées. (La fonction `sub_vect` est l'analogue pour les tableaux de la fonction `sub_string` des chaînes de caractères : elle extrait un sous-tableau d'une certaine longueur à partir d'un certain indice.)

La lecture et l'assemblage des programmes

Le module `lecture` fournit l'unique fonction `programme`, qui lit un programme de la pico-machine depuis un flux de caractères, l'assemble, puis fait résoudre les étiquettes par la fonction `extraire_code` qui renvoie le tableau d'instructions correspondant.

Fichier `lecture.mli`

```
#open "code";;
value programme: char stream -> instruction vect;;
```

La lecture n'est pas très complexe à comprendre, si ce n'est qu'on ne construit pas d'arbre de syntaxe abstraite : rien ne nous y oblige ici, puisque nous n'analyserons pas les programmes assemblés. On se contente donc d'assembler les instructions « au vol », dès leur lecture, en laissant évidemment non résolues les références en avant. C'est la phase d'extraction du code qui se chargera ensuite de cette résolution.

Pour l'analyse lexicale, nous réutilisons le générateur d'analyseurs lexicaux `lexuniv` introduit au chapitre 12, convenablement paramétré par la liste des mots-clés du langage d'assemblage.

Fichier `lecture.ml`

```
#open "code";;
#open "stockage";;
#open "lexuniv";;

let registre = function
  | [< 'MC "r"; 'Entier nbr >] -> nbr
  | [< 'MC "sp" >] -> sp
  | [< 'MC "ra" >] -> ra;;

let constante = function
  | [< 'Entier nbr >] -> nbr
  | [< 'Ident nom_étiq >] -> valeur_étiquette nom_étiq;;

let opérande = function
  | [< registre r >] -> Reg r
  | [< constante c >] -> Imm c;;

let rec instruction = function
  | [< opération op; reg_op_reg (r1, o, r2) >] ->
         assemble(Op(op, r1, o, r2))
  | [< test_inversé test; reg_op_reg (r1, o, r2) >] ->
         assemble(Op(test, r1, o, r2));
         assemble(Op(Seq, r2, Reg 0, r2))
  | [< 'MC "jmp"; opérande o; 'MC ","; registre r >] ->
         assemble(Jmp(o, r))
  | [< 'MC "braz"; registre r; 'MC ","; constante c >] ->
         assemble(Braz(r, c))
  | [< 'MC "branz"; registre r; 'MC ","; constante c >] ->
         assemble(Branz(r, c))
  | [< 'MC "scall"; 'Entier n >] -> assemble (Scall n)
  | [< 'MC "write" >] -> assemble (Scall 1)
  | [< 'MC "read" >] -> assemble (Scall 0)
  | [< 'MC "stop" >] -> assemble Stop

and reg_op_reg = function
  | [< registre r1; 'MC ","; opérande o; 'MC ","; registre r2 >] ->
     (r1, o, r2)

and opération = function
  | [< 'MC "load" >] -> Load    | [< 'MC "store" >] -> Store
  | [< 'MC "add" >]  -> Add     | [< 'MC "mult" >]  -> Mult
  | [< 'MC "sub" >]  -> Sub     | [< 'MC "div" >]   -> Div
  | [< 'MC "and" >]  -> And     | [< 'MC "or" >]    -> Or
  | [< 'MC "xor" >]  -> Xor     | [< 'MC "shl" >]   -> Shl
  | [< 'MC "shr" >]  -> Shr     | [< 'MC "slt" >]   -> Slt
  | [< 'MC "sle" >]  -> Sle     | [< 'MC "seq" >]   -> Seq

and test_inversé = function
  | [< 'MC "sgt" >] -> Sle
  | [< 'MC "sge" >] -> Slt
```

```
  | [< 'MC "sne" >] -> Seq;;

let définition_d'étiquette = function
  | [< 'Ident nom_étiq; 'MC ":" >] -> poser_étiquette nom_étiq;;

let rec instruction_étiq = function
  | [< définition_d'étiquette (); instruction_étiq () >] -> ()
  | [< instruction () >] -> ();;

let rec suite_d'instructions flux =
  match flux with
  | [< instruction_étiq () >] -> suite_d'instructions flux
  | [< >] -> ();;

let analyseur_lexical =
    construire_analyseur
      ["r"; "sp"; "ra"; "load"; "store"; "add"; "mult"; "sub"; "div";
       "and"; "or"; "xor"; "shl"; "shr"; "sgt"; "sge"; "sne";
       "slt"; "sle"; "seq"; "jmp"; "braz"; "branz";
       "scall"; "write"; "read"; "stop"; ","; ":"];;

let programme flux =
    initialise ();
    suite_d'instructions (analyseur_lexical flux);
    extraire_code ();;
```

L'assemblage d'un fichier complet

L'assemblage d'un fichier consiste simplement à le lire en mémoire, à l'assembler en résolvant les étiquettes, puis à écrire le tableau des instructions sur le fichier de sortie spécifié. La seule difficulté consiste à gérer les cas d'erreur.

Fichier `asm.ml`

```
let assemble_fichier nom_entrée nom_sortie =
    let entrée = open_in nom_entrée in
    let sortie = open_out_bin nom_sortie in
      try
        output_value sortie
          (lecture__programme (stream_of_channel entrée));
        close_in entrée;
        close_out sortie;
        0
      with exc ->
        close_in entrée;
        close_out sortie;
        sys__remove nom_sortie;
        match exc with
        | Parse_error | Parse_failure ->
            prerr_string
              "Erreur de syntaxe aux alentours du caractère numéro ";
            prerr_int (pos_in entrée);
```

```
            prerr_endline "";
            1
        | stockage__Erreur message ->
            prerr_string "Erreur d'assemblage: ";
            prerr_endline message;
            1
        | _ ->
            raise exc;;
```

La fonction principale se contente d'analyser ses arguments, puis si tout va bien, elle appelle la fonction `assemble_fichier` précédente.

Fichier `asm.ml`

```
exception Mauvais_arguments;;

if sys__interactive then () else
try
  if vect_length sys__command_line <> 3 then raise Mauvais_arguments;
  exit (assemble_fichier sys__command_line.(1) sys__command_line.(2))
with Mauvais_arguments ->
       prerr_endline
         "Usage: pico_asm <fichier assembleur> <fichier de code>";
       exit 2
   | sys__Sys_error message ->
       prerr_string "Erreur du système: "; prerr_endline message;
       exit 2;;
```

Comme pour la commande `pico_run` on compile, puis assemble les modules de l'assembleur, pour produire la commande `pico_asm` :

```
$ camlc -c stockage.mli
$ camlc -c stockage.ml
$ camlc -c lexuniv.mli
$ camlc -c lexuniv.ml
$ camlc -c lecture.mli
$ camlc -c lecture.ml
$ camlc -c asm.ml
$ camlc -o pico_asm code.zo stockage.zo lexuniv.zo lecture.zo asm.zo
```

Exemple

Nous exécutons la fonction factorielle à l'aide de la pico-machine. Il nous faut assembler le fichier `fact.asm` avec la commande `pico_asm`, puis charger les instructions en mémoire programme et lancer la pico-machine avec la commande `pico_run` :

```
$ pico_asm fact.asm fact.o
$ pico_run fact.o
10
3628800
```

Si l'on a compilé et chargé les modules depuis le système interactif, le même résultat s'obtient par les commandes `asm__assemble_fichier "fact.asm" "fact.o"` et `exec__exécute_fichier "fact.o" 4096`.

Adresse	Instruction	Assembleur source
		`# Le programme principal`
0:	`Scall 0`	`read`
4:	`Jmp (Imm 16, 31)`	`jmp fact, ra`
8:	`Scall 1`	`write`
12:	`Stop`	`stop`
		`# La fonction fact(N)`
16:	`Braz (1, 60)`	`fact: braz r 1, fact0`
20:	`Op (Sub, 30, Imm 8, 30)`	`sub sp, 8, sp`
24:	`Op (Store, 30, Imm 0, 31)`	`store sp, 0, ra`
28:	`Op (Store, 30, Imm 1, 1)`	`store sp, 4, r 1`
32:	`Op (Sub, 1, Imm 1, 1)`	`sub r 1, 1, r 1`
36:	`Jmp (Imm 16, 31)`	`jmp fact, ra`
40:	`Op (Load, 30, Imm 1, 2)`	`load sp, 4, r 2`
44:	`Op (Mult, 1, Reg 2, 1)`	`mult r 1, r 2, r 1`
48:	`Op (Load, 30, Imm 0, 31)`	`load sp, 0, ra`
52:	`Op (Add, 30, Imm 8, 30)`	`add sp, 2, sp`
56:	`Jmp (Reg 31, 0)`	`jmp ra, r 0`
60:	`Op (Add, 0, Imm 1, 1)`	`fact0: add r 0, 1, r 1`
64:	`Jmp (Reg 31, 0)`	`jmp ra, r 0`

Figure 14.3: Résultat de l'assemblage du fichier `fact.asm`

À titre d'exemple, nous donnons figure 14.3 le code assemblé par la commande `pico_asm` pour le fichier `fact.asm` (page 268), en faisant figurer, en regard de chaque instruction assemblée, le code source correspondant du fichier. On constate sur cet exemple que les étiquettes ont été résolues correctement et que le registre `sp` est bien expansé en son numéro absolu.

14.4 Pour aller plus loin

Le modèle de pico-processeur que nous avons décrit n'est pas complètement réaliste : pour simplifier, nous n'avons pas rendu compte d'un certain nombre de traits des « vrais » processeurs, traits qui sont des conséquences directes de l'architecture interne de ces processeurs. Par exemple, dans le processeur MIPS R3000 dont nous nous sommes inspirés, certaines instructions prennent effet « à retardement » : un branchement, par exemple, n'est pas exécuté immédiatement ; le processeur exécute systématiquement l'instruction qui suit le branchement avant de se dérouter effectivement à l'endroit indiqué. Le pico-processeur ne simule pas ce fait. De même, nous n'avons pas essayé de simuler fidèlement le temps d'exécution des programmes : même si, sur un processeur RISC, la plupart des instructions s'exécutent en un cycle d'horloge, certaines instructions arithmétiques (multiplication et division) prennent généralement plus de temps ; pis, les accès à la mémoire prennent des temps très variables suivant qu'on tombe dans la mémoire cache de niveau 1 (2 à 3 cycles, typiquement), celle de niveau 2 (10 à 30 cycles), dans la mémoire principale (40 à 100 cycles), ou dans la mémoire virtuelle (des millions de cycles). Par conséquent, le programme décrit dans ce chapitre est davantage un interpréteur d'un langage d'assemblage raisonnablement réaliste qu'un simulateur

fidèle d'un processeur réel. Simuler fidèlement un processeur réel est un exercice de programmation intéressant, quoique difficile.

Bibliographie

Pour une introduction progressive et très complète aux architectures de processeurs, on lira avec profit *Architecture des ordinateurs : approche quantitative*, de Hennessy et Patterson (International Thompson Publishing).

15
Compilation de mini-Pascal

Un mini-Pascal pour une pico-machine, mais un programme respectable quand même.

Une fois maîtrisées les techniques de l'analyse syntaxique et de la manipulation d'arbres de syntaxe abstraite, il est naturel de les appliquer à l'implémentation en Caml de véritables langages de programmation. Dans ce chapitre, nous écrivons un compilateur pour un petit langage impératif dans le style de Pascal, mais très simplifié. Le code produit par le compilateur est exécutable par le simulateur du chapitre 14. C'est l'occasion de montrer l'architecture générale d'un compilateur et d'introduire quelques algorithmes classiques de génération de code. Ce chapitre est également un bon exemple de structuration d'un programme assez complexe.

15.1 Syntaxe abstraite, syntaxe concrète

Le langage auquel nous allons nous intéresser est un sous-ensemble de Pascal. Les seuls types de données sont les entiers, les booléens et les tableaux à indices entiers. Au niveau des instructions, certains types de boucles ont été omis. On dispose de procédures et de fonctions, mais elles ne peuvent pas être locales à une autre procédure ou fonction. Les paramètres sont passés par valeur pour les entiers et les booléens et par référence pour les tableaux. Dernière différence majeure par rapport à Pascal : les procédures et les fonctions sont considérées comme mutuellement récursives ; on peut donc appeler une procédure avant de l'avoir définie (comme en Modula-2).

À titre d'exemple, voici deux programmes mini-Pascal qui calculent la fonction de Fibonacci, de manière plus ou moins naïve.

Fichier `fib1.pas`

```
program fibonacci;
var n: integer;
function fib(n: integer): integer;
  begin if n < 2 then fib := 1 else fib := fib(n - 1) + fib(n - 2) end;
begin
  read(n); write(fib(n))
```

```
end
```

Fichier `fib2.pas`

```
program fibonacci;
var fib: array [0 .. 100] of integer;
var n: integer;
var i: integer;
begin
  read(n);
  fib[0] := 1; fib[1] := 1; i := 2;
  while i <= n do begin
    fib[i] := fib[i - 1] + fib[i - 2]; i := i + 1
  end;
  write(fib[n])
end
```

La syntaxe abstraite (c'est-à-dire la représentation interne) des programmes écrits dans ce langage est structurée en plusieurs niveaux, correspondant chacun à un type concret Caml :

Niveau	Type Caml	Exemple
Constantes	`constante`	`true`
Expressions	`expression`	`x+1`
Instructions	`instruction`	`x:=x+1`
Expressions de type	`expr_type`	`array [1..10] of integer`
Déclarations de procédures	`décl_proc`	`procedure p(x:int)...`
Déclarations de fonctions	`décl_fonc`	`function f(x:int):int...`
Programmes	`programme`	`program prog; ...`

Ces types concrets sont définis dans l'interface du module `syntaxe`.

Fichier `syntaxe.mli`

```
type constante =
   | Entière of int
   | Booléenne of bool;;

type expr_type =
   | Integer                          (* le type des entiers *)
   | Boolean                          (* le type des booléens *)
   | Array of int * int * expr_type;; (* le type des tableaux *)
                          (* (les deux "int" sont les bornes) *)
type expression =
   | Constante of constante
   | Variable of string
   | Application of string * expression list
   | Op_unaire of string * expression
   | Op_binaire of string * expression * expression
   | Accès_tableau of expression * expression;;

type instruction =
   | Affectation_var of string * expression
```

```
   | Affectation_tableau of expression * expression * expression
   | Appel of string * expression list   (* appel de procédure *)
   | If of expression * instruction * instruction
   | While of expression * instruction
   | Write of expression
   | Read of string
   | Bloc of instruction list;;          (* bloc begin ... end *)

type décl_proc =
  { proc_paramètres: (string * expr_type) list;
    proc_variables: (string * expr_type) list;
    proc_corps: instruction }
and décl_fonc =
  { fonc_paramètres: (string * expr_type) list;
    fonc_type_résultat: expr_type;
    fonc_variables: (string * expr_type) list;
    fonc_corps: instruction };;

type programme =
  { prog_variables: (string * expr_type) list;
    prog_procédures: (string * décl_proc) list;
    prog_fonctions: (string * décl_fonc) list;
    prog_corps: instruction };;

value lire_programme : char stream -> programme;;
```

L'implémentation du module `syntaxe` est entièrement consacrée à l'analyseur syntaxique (la fonction `lire_programme` déclarée ci-dessus). Nous réutilisons l'analyseur lexical « universel » `lexuniv` introduit au chapitre 12 pour la lecture des propositions et utilisé également au chapitre 14 pour l'assembleur de la pico-machine.

Fichier `syntaxe.ml`

```
#open "lexuniv";;
let analyseur_lexical = construire_analyseur
  ["false";"true";"("; ","; ")"; "["; "]"; "not"; "*"; "/"; "-"; "+";
   "="; "<>"; "<"; ">"; "<="; ">="; "and"; "or"; "if"; "then"; "else";
   "while"; "do"; "write"; "read"; "begin"; ";"; "end"; ":=";
   "integer"; "boolean"; "array"; "of"; ".."; "var"; ":";
   "procedure"; "function"; "program"];;
```

L'analyseur lexical s'obtient par application partielle de la fonction `construire_analyseur` à la liste des mots-clés. Viennent ensuite deux puissantes fonctionnelles d'analyse syntaxique, l'une pour analyser des listes, l'autre pour analyser des applications d'opérateurs infixes. Voici la fonctionnelle d'analyse des listes.

Fichier `syntaxe.ml`

```
let lire_liste lire_élément séparateur =
  let rec lire_reste = function
    | [< (stream_check
           (function lexème -> lexème = MC séparateur)) sép;
         lire_élément elt;
```

```
        lire_reste reste >] -> elt :: reste
    | [< >] -> [] in
  function [< lire_élément elt; lire_reste reste >] -> elt :: reste
        | [< >] -> [];;
```

La fonctionnelle `lire_liste` prend en argument un analyseur `lire_élément` et une chaîne `séparateur` et renvoie un analyseur qui reconnaît les listes d'éléments reconnus par `lire_éléments`, séparés par des occurrences de la chaîne `séparateur`. Par exemple,

```
lire_liste (function [< 'Entier n >] -> n) ","
```

est une fonction de type `lexème stream -> int list` qui reconnaît les listes d'entiers séparés par des virgules. En appliquant cette fonction au flux `[< 'Entier 1; 'MC ","; 'Entier 2 >]`, on obtient la liste `[1;2]`.

La fonction `stream_check` employée dans `lire_reste` permet de filtrer les caractères qui vérifient une certaine condition. De manière générale, un motif de la forme `[< (stream_check p) c; ... >]` est sélectionné si la fonction `p` appliquée au premier élément du flux renvoie `true`. Le premier élément du flux est alors lié à la variable `c` et le filtrage continue comme d'habitude. Dans le cas de `lire_reste`, le motif

```
[< (stream_check (function lexème -> lexème = MC séparateur)) sép >]
```

filtre donc les mots-clés dont le texte est identique à la chaîne `séparateur`.

L'autre fonctionnelle sert à analyser les applications d'opérateurs binaires infixes, comme `+` ou `and`.

Fichier `syntaxe.ml`

```
let est_un_opérateur opérateurs = function
  | MC op -> mem op opérateurs
  | _     -> false;;

let lire_opérateur opérateurs = function
  | [< (stream_check (est_un_opérateur opérateurs)) (MC op) >] -> op;;

let lire_opération lire_base opérateurs =
  let rec lire_reste e1 = function
  | [< (lire_opérateur opérateurs) op;
       lire_base e2;
       (lire_reste (Op_binaire(op, e1, e2))) e >] -> e
  | [< >] -> e1 in
 function [< lire_base e1; (lire_reste e1) e >] -> e;;
```

La fonction `lire_opérateur` prend en argument une liste d'opérateurs représentés par des chaînes de caractères, comme `["+";"-"]`, et rend un analyseur reconnaissant les flux qui commencent par un de ces opérateurs.

La fonctionnelle `lire_opération` prend en arguments un analyseur pour les expressions simples et une liste d'opérateurs et reconnaît les applications de ces opérateurs à des expressions simples. Par exemple,

```
lire_opération (function [< 'Entier n >] -> Constante(Entière n))
               ["+";"-"]
```

renvoie un analyseur de type `lexème stream -> expression` reconnaissant des expressions comme `1+2-3`. En appliquant cet analyseur au flux

```
[< 'Entier 1; 'MC "+"; 'Entier 2; 'MC "-"; 'Entier 3 >]
```

on obtient l'expression

```
Op_binaire("-",
  Op_binaire("+", Constante(Entière 1), Constante(Entière 2)),
  Constante(Entière 3))
```

Pour construire l'arbre de syntaxe, on considère que les opérateurs sont associatifs à gauche ; autrement dit, `1+2-3` est lue comme `(1+2)-3` et non pas comme `1+(2-3)`.

Le reste de l'analyseur syntaxique est plus simple à comprendre. Pour les expressions, nous introduisons un certain nombre d'analyseurs intermédiaires, afin de respecter les priorités usuelles entre opérateurs : `*` est plus prioritaire que `+`, `and` est plus prioritaire que `or`, etc.

Fichier `syntaxe.ml`

```
let rec lire_expr0 flux =
  match flux with
  | [< 'Entier n >] -> Constante(Entière n)
  | [< 'MC "false" >] -> Constante(Booléenne false)
  | [< 'MC "true" >] -> Constante(Booléenne true)
  | [< 'Ident nom >] ->
      begin match flux with
      | [< 'MC "("; (lire_liste lire_expr ",") el; 'MC ")">] ->
                  Application(nom, el)
      | [< >] -> Variable nom
      end
  | [< 'MC "("; lire_expr e; 'MC ")" >] -> e

and lire_expr1 flux =
  match flux with
    [< lire_expr0 e1 >] ->
      match flux with
      | [< 'MC "["; lire_expr e2; 'MC "]" >] -> Accès_tableau(e1,e2)
      | [< >] -> e1

and lire_expr2 = function
  | [< 'MC "-"; lire_expr1 e >] -> Op_unaire("-", e)
  | [< 'MC "not"; lire_expr1 e >] -> Op_unaire("not", e)
  | [< lire_expr1 e >] -> e

and lire_expr3 flux =
  lire_opération lire_expr2 ["*"; "/"] flux
and lire_expr4 flux =
  lire_opération lire_expr3 ["+"; "-"] flux
and lire_expr5 flux =
  lire_opération lire_expr4 ["="; "<>"; "<"; ">"; "<="; ">="] flux
and lire_expr6 flux =
  lire_opération lire_expr5 ["and"] flux
and lire_expr flux =
  lire_opération lire_expr6 ["or"] flux;;
```

Viennent ensuite des analyseurs pour les instructions, les types, les déclarations

de variables, les déclarations de fonctions, les déclarations de procédures et enfin les programmes.

Fichier `syntaxe.ml`

```
let rec lire_instr flux =
  match flux with
  | [< 'MC "if"; lire_expr e1; 'MC "then"; lire_instr i2 >] ->
      begin match flux with
      | [< 'MC "else"; lire_instr i3 >] -> If(e1, i2, i3)
      | [< >] -> If(e1, i2, Bloc [])
      end
  | [< 'MC "while"; lire_expr e1; 'MC "do"; lire_instr i2 >] ->
      While(e1, i2)
  | [< 'MC "write"; 'MC "("; lire_expr e; 'MC ")" >] ->
      Write e
  | [< 'MC "read"; 'MC "("; 'Ident nom; 'MC ")" >] ->
      Read nom
  | [< 'MC "begin"; (lire_liste lire_instr ";") il; 'MC "end" >] ->
      Bloc il
  | [< lire_expr e >] ->
      match e with
      | Application(nom, el) ->
          Appel(nom, el)
      | Variable nom ->
          begin match flux with
          | [< 'MC ":="; lire_expr e >] ->
              Affectation_var(nom, e)
          end
      | Accès_tableau(e1, e2) ->
          begin match flux with
          | [< 'MC ":="; lire_expr e3 >] ->
              Affectation_tableau(e1, e2, e3)
          end
      | _ -> raise Parse_error;;

let rec lire_type = function
  | [< 'MC "integer" >] -> Integer
  | [< 'MC "boolean" >] -> Boolean
  | [< 'MC "array"; 'MC "["; 'Entier bas; 'MC ".."; 'Entier haut;
       'MC "]"; 'MC "of"; lire_type ty >] -> Array(bas, haut, ty);;

let rec lire_variables = function
  | [< 'MC "var"; 'Ident nom; 'MC ":"; lire_type ty; 'MC ";";
       lire_variables reste >] -> (nom,ty)::reste
  | [< >] -> [];;

let lire_un_paramètre = function
  | [< 'Ident nom; 'MC ":"; lire_type ty >] -> (nom,ty);;

let lire_paramètres = function
  | [< 'MC "(";
       (lire_liste lire_un_paramètre ",") paramètres;
```

```
       'MC ")" >] -> paramètres;;

let lire_procédure = function
  | [< 'MC "procedure"; 'Ident nom; lire_paramètres p; 'MC ";";
       lire_variables v; lire_instr i; 'MC ";" >] ->
       (nom, {proc_paramètres=p; proc_variables=v; proc_corps=i});;

let lire_fonction = function
  | [< 'MC "function"; 'Ident nom; lire_paramètres p; 'MC ":";
       lire_type ty; 'MC ";"; lire_variables v;
       lire_instr i; 'MC ";" >] ->
         (nom, {fonc_paramètres=p; fonc_type_résultat=ty;
                fonc_variables=v; fonc_corps=i});;

let rec lire_proc_fonc = function
  | [< lire_procédure proc; lire_proc_fonc (procs, foncs) >] ->
      (proc::procs, foncs)
  | [< lire_fonction fonc;  lire_proc_fonc (procs, foncs) >] ->
       (procs, fonc::foncs)
  | [< >] -> ([], []);;

let lire_prog = function
  | [< 'MC "program"; 'Ident nom_du_programme; 'MC ";";
       lire_variables v; lire_proc_fonc (p,f); lire_instr i >] ->
    { prog_variables=v; prog_procédures=p;
      prog_fonctions=f; prog_corps=i };;

let lire_programme flux = lire_prog (analyseur_lexical flux);;
```

15.2 Typage

Nous programmons maintenant un vérificateur de types pour les programmes mini-Pascal. Le but premier du vérificateur de types est de garantir l'absence d'incohérences entre les types des objets manipulés par le programme ; par exemple, l'addition d'un entier et d'un tableau doit être rejetée. Un but secondaire est de détecter certaines opérations comme l'affectation entre tableaux ou le renvoi d'un tableau comme résultat d'une fonction, opérations que nous avons décidé d'interdire pour faciliter la compilation du langage.

De manière générale, les erreurs de types peuvent être détectées ou bien au moment de l'exécution (*typage dynamique*), ou bien avant l'exécution, par une analyse préalable du programme (*typage statique*). Le typage statique offre plusieurs avantages par rapport au typage dynamique. Tout d'abord, il détecte les erreurs de types dans toutes les branches du programme, même celles qui ne sont pas toujours exécutées. De plus, dans le cadre d'un compilateur, le typage statique nous autorise à produire du code ne contenant aucun test sur le type des objets, puisqu'il garantit que les objets manipulés pendant l'exécution seront forcément du bon type pour les opérations effectuées dessus. Au contraire, pour faire du typage dynamique, il faudrait produire du code pour vérifier les types à l'exécution, ce qui complique la compilation et ralentit l'exécution.

Voici l'interface du module `typage` fournissant la fonction de vérification des types.

Fichier `typage.mli`

```
#open "syntaxe";;

type erreur_de_type =
   | Indéfini of string      (* variable utilisée mais non définie *)
   | Conflit of string * expr_type * expr_type (* conflit de types *)
   | Arité of string * int * int     (* mauvais nombre d'arguments *)
   | Tableau_attendu             (* [..] appliqué à un non-tableau *)
   | Tableau_interdit of string;;   (* tableau renvoyé en résultat *)

exception Erreur_typage of erreur_de_type;;

value type_programme: programme -> unit
  and affiche_erreur: erreur_de_type -> unit
  and type_op_unaire: string -> expr_type * expr_type
  and type_op_binaire: string -> expr_type * expr_type * expr_type;;
```

La fonction `type_programme` signale les erreurs de typage en déclenchant alors l'exception `Erreur_typage` avec pour argument une description de l'erreur. L'erreur peut ensuite être imprimée par la fonction `affiche_erreur`.

Environnements de typage

Pour typer une expression ou une instruction, il est nécessaire de connaître les types des variables, des fonctions et des procédures mentionnées dans cette expression ou cette instruction. L'environnement de typage est une structure de données qui associe aux noms de variables leur type courant, c'est-à-dire le type de la déclaration la plus récente de chaque variable. Il associe également aux noms de procédures et de fonctions leurs déclarations (noms des paramètres, corps de la procédure, etc.).

Le module `envir` fournit le type abstrait `env` des environnements et les opérations de base sur ce type. Pour pouvoir réutiliser le module `envir`, nous allons paramétrer le type `env` par le type des informations associées aux variables. Le type fourni par le module `envir` est donc `'a env`, où `'a` est le type des informations associées aux variables. Dans le vérificateur, nous associons des expressions de types aux variables, et utilisons donc des environnements de type `expr_type env`. Dans le compilateur (section 15.3), ce sont des informations de compilation que nous associerons aux variables.

Fichier `envir.mli`

```
#open "syntaxe";;
type 'a env;;
value environnement_initial:
      (string * décl_proc) list -> (string * décl_fonc) list -> 'a env
  and ajoute_variable: string -> 'a -> 'a env -> 'a env
  and cherche_variable: string -> 'a env -> 'a
  and cherche_fonction: string -> 'a env -> décl_fonc
  and cherche_procédure: string -> 'a env -> décl_proc;;
exception Pas_trouvé of string;;
```

En voici une implémentation simple, à base de listes d'associations.

Fichier `envir.ml`

```
#open "syntaxe";;
#open "interp";;
type 'a env =
  { vars: (string * 'a) list;
    procs: (string * décl_proc) list;
    foncs: (string * décl_fonc) list };;

let environnement_initial p f =
  { vars=[]; procs=p; foncs=f };;

let ajoute_variable nom info env =
  { vars=(nom,info)::env.vars; procs=env.procs; foncs=env.foncs };;

let cherche nom liste =
  try assoc nom liste with Not_found -> raise(Pas_trouvé nom);;

let cherche_variable nom env = cherche nom env.vars
and cherche_fonction nom env = cherche nom env.foncs
and cherche_procédure nom env = cherche nom env.procs;;
```

Typage des expressions

L'implémentation du module `typage` commence par trois fonctions élémentaires de vérification sur les types.

Fichier `typage.ml`

```
#open "syntaxe";;
#open "envir";;

let vérifie_type message type_attendu type_réel =
  if type_attendu <> type_réel then
    raise(Erreur_typage(Conflit(message, type_attendu, type_réel)));;

let vérifie_tableau = function
  | Array(inf, sup, éléments) -> éléments
  | _ -> raise(Erreur_typage(Tableau_attendu));;

let vérifie_non_tableau message = function
  | Array(inf, sup, éléments) ->
      raise(Erreur_typage(Tableau_interdit message))
  | _ -> ();;
```

Passons ensuite à la fonction `type_expr`, qui calcule le type d'une expression, ou déclenche l'exception `Erreur_typage` si l'expression est mal typée. Cette fonction prend en argument un environnement qui fait correspondre aux variables leur type, aux procédures et aux fonctions leur déclaration.

Fichier `typage.ml`

```
let rec type_expr env = function
  | Constante(Entière n) -> Integer
```

```
  | Constante(Booléenne b) -> Boolean
  | Variable nom_var ->
      cherche_variable nom_var env
  | Application(nom_fonc, args) ->
      let fonc = cherche_fonction nom_fonc env in
      type_application env nom_fonc fonc.fonc_paramètres args;
      fonc.fonc_type_résultat
  | Op_unaire(op, arg) ->
      let (type_arg, type_res) = type_op_unaire op in
      vérifie_type ("l'argument de " ^ op)
                   type_arg (type_expr env arg);
      type_res
  | Op_binaire(op, arg1, arg2) ->
      let (type_arg1, type_arg2, type_res) = type_op_binaire op in
      vérifie_type ("le premier argument de " ^ op)
                   type_arg1 (type_expr env arg1);
      vérifie_type ("le deuxième argument de " ^ op)
                   type_arg2 (type_expr env arg2);
      type_res
  | Accès_tableau(expr1, expr2) ->
      let type_éléments = vérifie_tableau (type_expr env expr1) in
      vérifie_type "l'indice de tableau"
                   Integer (type_expr env expr2);
      type_éléments

and type_application env nom paramètres arguments =
  let nbr_paramètres = list_length paramètres
  and nbr_arguments = list_length arguments in
  if nbr_paramètres <> nbr_arguments then
    raise(Erreur_typage(Arité(nom, nbr_paramètres, nbr_arguments)));
  let type_paramètre (nom_param, type_param) argument =
    vérifie_type ("le paramètre " ^ nom_param ^ " de " ^ nom)
                 type_param (type_expr env argument) in
  do_list2 type_paramètre paramètres arguments

and type_op_unaire = function
  | "-" -> (Integer, Integer)
  | "not" -> (Boolean, Boolean)

and type_op_binaire = function
  | "*" | "/" | "+" | "-" -> (Integer,Integer,Integer)
  | "=" | "<>" | "<" | ">" | "<=" | ">=" -> (Integer,Integer,Boolean)
  | "and" | "or" -> (Boolean,Boolean,Boolean);;
```

Typage des instructions

L'étape suivante consiste à typer les instructions. Au contraire de `type_expr`, la fonction `type_instr` ne renvoie rien : il n'y a pas de type à calculer, seulement des types à vérifier.

Fichier `typage.ml`

```
let rec type_instr env = function
  | Affectation_var(nom_var, expr) ->
      let type_var = cherche_variable nom_var env in
      vérifie_non_tableau ("affectation de " ^ nom_var) type_var;
      vérifie_type ("la variable " ^ nom_var)
                   type_var (type_expr env expr)
  | Affectation_tableau(expr1, expr2, expr3) ->
      let type_éléments = vérifie_tableau (type_expr env expr1) in
      vérifie_non_tableau "affectation de tableau" type_éléments;
      vérifie_type "l'indice de tableau"
                   Integer (type_expr env expr2);
      vérifie_type "affectation de tableau"
                   type_éléments (type_expr env expr3)
  | Appel(nom_proc, args) ->
      let proc = cherche_procédure nom_proc env in
      type_application env nom_proc proc.proc_paramètres args
  | If(condition, branche_oui, branche_non) ->
      vérifie_type "la condition de IF"
                   Boolean (type_expr env condition);
      type_instr env branche_oui;
      type_instr env branche_non
  | While(condition, corps) ->
      vérifie_type "la condition de WHILE"
                   Boolean (type_expr env condition);
      type_instr env corps
  | Write expr ->
      vérifie_type "l'argument de WRITE"
                   Integer (type_expr env expr)
  | Read nom_var ->
      vérifie_type "l'argument de READ"
                   Integer (cherche_variable nom_var env)
  | Bloc liste ->
      do_list (type_instr env) liste;;
```

Typage d'un programme

Les fonctions de typage d'une déclaration de fonction ou de procédure ajoutent dans l'environnement les types déclarés pour les paramètres et les variables locales, puis vérifient dans cet environnement le typage du corps de la fonction ou de la procédure.

Fichier `typage.ml`

```
let ajoute_var (nom, typ) env = ajoute_variable nom typ env;;

let type_procédure env_global (nom, décl) =
  let env =
    list_it ajoute_var
            (décl.proc_variables @ décl.proc_paramètres)
            env_global in
  type_instr env décl.proc_corps;;
```

```
let type_fonction env_global (nom, décl) =
  vérifie_non_tableau
    ("passage comme résultat de la fonction " ^ nom)
    décl.fonc_type_résultat;
  let env =
    list_it ajoute_var
            ((nom, décl.fonc_type_résultat) ::
              décl.fonc_variables @ décl.fonc_paramètres)
            env_global in
  type_instr env décl.fonc_corps;;
```

Enfin, le typage d'un programme tout entier consiste à construire un environnement de typage global, correspondant aux déclarations de variables globales, de fonctions et de procédures, puis à vérifier les types dans les fonctions, les procédures et le corps du programme.

Fichier `typage.ml`

```
let type_programme prog =
  let env_global =
    list_it ajoute_var prog.prog_variables
            (environnement_initial prog.prog_procédures
                                   prog.prog_fonctions) in
  try
    do_list (type_procédure env_global) prog.prog_procédures;
    do_list (type_fonction env_global) prog.prog_fonctions;
    type_instr env_global prog.prog_corps
  with Pas_trouvé nom ->
    raise(Erreur_typage(Indéfini nom));;
```

Traitement des erreurs de typage

Il reste à afficher les messages d'erreur. On utilise pour cela les fonctions de bibliothèque `prerr_int` et `prerr_string` de préférence à `print_string` et `print_int`, car les premières affichent sur la sortie d'erreur standard et non sur la sortie standard comme les secondes.

Fichier `typage.ml`

```
let rec affiche_type = function
  | Integer -> prerr_string "integer"
  | Boolean -> prerr_string "boolean"
  | Array(inf, sup, ty) ->
      prerr_string "array ["; prerr_int inf; prerr_string "..";
      prerr_int sup; prerr_string "] of "; affiche_type ty;;

let affiche_erreur = function
  | Indéfini nom ->
      prerr_string "Nom inconnu: "; prerr_string nom;
      prerr_endline "."
  | Conflit(message, type_attendu, type_réel) ->
      prerr_string "Conflit de types: "; prerr_string message;
      prerr_string " devrait avoir le type ";
```

```
      affiche_type type_attendu;
      prerr_string " mais a le type "; affiche_type type_réel;
      prerr_endline "."
  | Arité(nom, nbr_paramètres, nbr_arguments) ->
      prerr_string "Mauvais nombre d'arguments: "; prerr_string nom;
      prerr_string " attend "; prerr_int nbr_paramètres;
      prerr_string " paramètre(s), mais est appelée avec ";
      prerr_int nbr_arguments; prerr_endline " argument(s)."
  | Tableau_attendu ->
      prerr_endline "Accès dans un objet qui n'est pas un tableau."
  | Tableau_interdit message ->
      prerr_string "Opération interdite sur les tableaux: ";
      prerr_string message; prerr_endline ".";;
```

Les textes des messages d'erreur sont assez descriptifs. Il leur manque néanmoins une information fort utile : le numéro de la ligne où l'erreur s'est produite. Cette information ne figure pas dans l'arbre de syntaxe abstraite. Pour produire de meilleurs messages d'erreur, une première possibilité est de faire le typage en même temps que l'analyse syntaxique, auquel cas on irait simplement consulter une variable globale contenant le numéro de la ligne en cours d'analyse, variable tenue à jour par l'analyseur lexical. Cette solution est simple, mais complique la structure du compilateur. Une autre solution, plus générale, est de travailler sur un arbre de syntaxe abstraite annoté par les numéros de lignes correspondants dans le texte source. Par exemple, pour annoter chaque instruction, il faudrait déclarer le type `instruction` du module `syntaxe` comme suit :

```
type instruction =
  { description: descr_instruction;
    ligne: int }
and descr_instruction =
  | Affectation_var of string * expression
  | If of expression * instruction * instruction
...
```

Chaque nœud du type `instruction` dans l'arbre de syntaxe abstraite est ainsi annoté par un entier : le numéro de ligne. Nous n'avons pas utilisé cette technique dans ce chapitre, car elle alourdit désagréablement toutes les fonctions qui opèrent sur l'arbre de syntaxe abstraite. C'est cependant une technique très générale, qui peut servir pour bien d'autres types d'annotations en plus des numéros de lignes : types des expressions, informations de compilation, informations de mise au point (*debugging*).

15.3 Compilation

Cette section présente un compilateur pour le langage mini-Pascal, produisant du code pour le pico-processeur décrit dans le chapitre 14. Le compilateur se présente sous la forme d'un module `compil`, dont l'interface est fort simple (au contraire de l'implémentation ...).

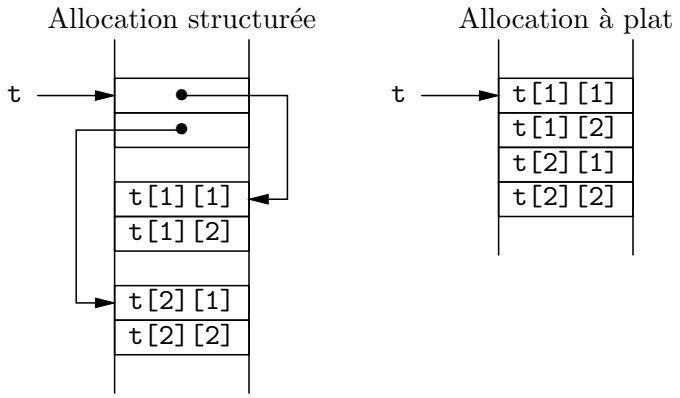
Fichier `compil.mli`

```
value compile_programme: syntaxe__programme -> unit;;
```

La fonction `compile_programme` écrit directement sur la sortie standard le code assembleur pour le programme donné en argument.

Représentation des données

Commençons par définir comment les types de données mini-Pascal sont représentés en machine. Les entiers (type `integer`) sont directement représentés par les entiers de la machine. Pour les booléens (type `boolean`), nous représentons `false` par l'entier 0 et `true` par l'entier 1. Un tableau (type `array`) est stocké dans un certain nombre de mots mémoire contigus et représenté par l'adresse mémoire du premier élément. Dans le cas d'un tableau de tableaux (c'est-à-dire un tableau multidimensionnel), deux approches sont possibles : l'allocation structurée ou l'allocation à plat. Avec l'allocation structurée, une valeur de type `array [1..2] of array [1..2] of integer` est représentée par un tableau de deux pointeurs vers deux tableaux de deux entiers. Avec l'allocation à plat, une valeur du même type est un bloc mémoire de quatre mots, contenant les deux tableaux de deux entiers mis côte à côte.



Nous allons utiliser l'allocation à plat, qui est plus compacte et plus efficace à l'accès que l'allocation structurée, bien qu'elle complique légèrement la compilation. En particulier, pour calculer le décalage entre le début d'un tableau et l'élément auquel on veut accéder, il faut connaître à la compilation la taille des éléments du tableau. Heureusement, cette taille se déduit facilement du type des éléments du tableau, qui est une information connue à la compilation. La fonction `taille_du_type` ci-dessous calcule le nombre d'octets occupés par la représentation d'un objet du type donné.

Fichier `compil.ml`

```
#open "syntaxe";;
#open "envir";;
#open "printf";;

let taille_du_mot = 4;;            (* un mot = quatre octets *)

let rec taille_du_type = function
  | Integer | Boolean -> taille_du_mot
```

```
  | Array(inf, sup, ty) -> (sup - inf + 1) * taille_du_type ty;;
```

La fonction `val_const` traduit une constante en l'entier qui la représente.

Fichier `compil.ml`

```
let val_const = function
  | Entière n -> n
  | Booléenne b -> if b then 1 else 0;;
```

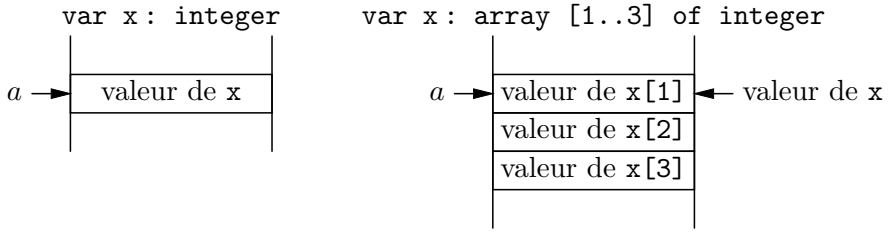
Environnements de compilation

Les fonctions de compilation prennent en argument un environnement qui indique à quels emplacements mémoire se trouvent les variables. On réutilise la structure générique d'environnement fournie par le module `envir`. Les données associées aux noms de variables sont du type `info_variable` défini ci-dessous.

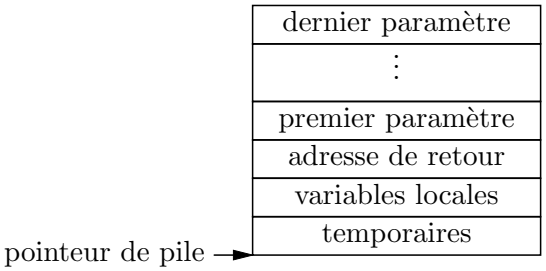
Fichier `compil.ml`

```
type info_variable =
  { typ: expr_type;
    emplacement: emplacement_variable }
and emplacement_variable =
  | Global_indirect of int
  | Global_direct of int
  | Local_indirect of int
  | Local_direct of int;;
```

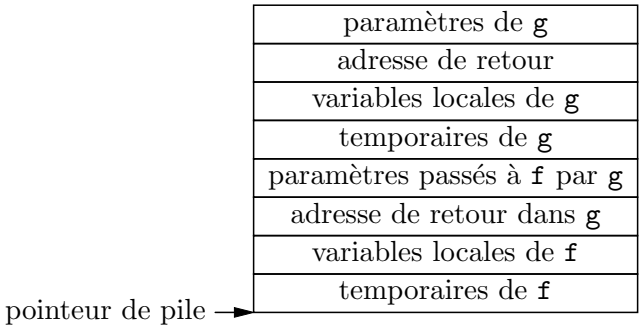
Le compilateur attribue une adresse absolue à chaque variable globale. Cette adresse est l'argument des constructeurs `Global_direct` et `Global_indirect`. La distinction entre ces deux constructeurs est la suivante : si la variable est un entier ou un booléen, l'entier associé est l'adresse d'un mot qui contient la valeur courante de `x`. En revanche, si la variable est un tableau, l'entier associé est l'adresse du premier mot du bloc mémoire correspondant ; la variable s'évalue donc en l'adresse elle-même et non pas en le mot contenu à cette adresse. L'emplacement `Global_indirect` a correspond au premier cas (une indirection à partir de a est nécessaire) ; l'emplacement `Global_direct` a correspond au deuxième cas (pas d'indirection à partir de a).



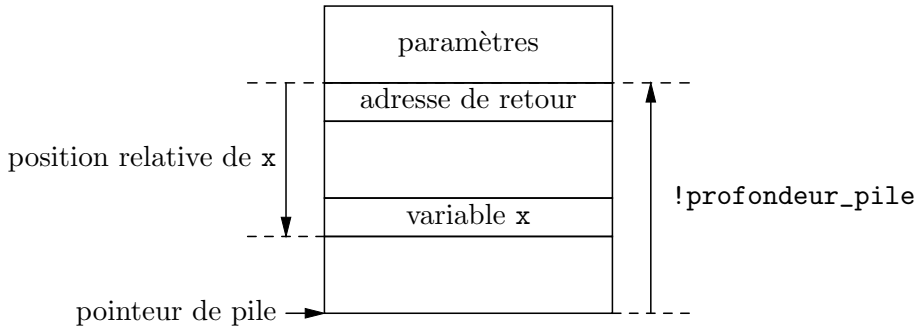
Les variables locales des fonctions et des procédures, ainsi que leurs paramètres, sont stockées sur la pile. Chaque fonction ou procédure s'alloue, quand elle est appelée, un bloc dans la pile appelé bloc d'activation (*activation record* ou *stack frame*, dans la littérature). Le bloc d'activation contient les paramètres, les variables locales et l'adresse de retour à l'appelant. Les blocs d'activation ont la structure suivante (la pile croît vers le bas) :



Les blocs d'activation s'empilent les uns derrière les autres au gré des appels de fonctions et de procédures. Par exemple, si `g` appelle `f`, voici l'état de la pile pendant que `f` s'exécute :



L'entier argument des constructeurs `Local_indirect` et `Local_direct` représente la position relative de la variable locale dans le bloc d'activation. Le point de repère est l'adresse du mot contenant le premier paramètre. Les positions positives correspondent aux variables locales ; les positions négatives, aux paramètres. Comme pour les valeurs globales, `Local_indirect` signifie qu'il faut faire une indirection à cette adresse pour obtenir la valeur de la variable, alors que `Local_direct` signifie que cette adresse est la valeur de la variable.



L'accès aux variables locales se fait par adressage relatif au pointeur de pile (le registre numéro 30, aussi appelé `sp`). Le compilateur garde trace du décalage entre le pointeur de pile et le mot contenant l'adresse de retour dans le bloc d'activation courant, dans la variable `profondeur_pile`. Cette variable augmente lorsqu'on alloue de nouveaux temporaires et diminue quand on les libère.

Fichier `compil.ml`

```
let profondeur_pile = ref 0;;
```

```
let réserve_pile n =
  printf "sub sp, %d, sp\n" (n * taille_du_mot);
  profondeur_pile := !profondeur_pile + n * taille_du_mot

and libère_pile n =
  printf "add sp, %d, sp\n" (n * taille_du_mot);
  profondeur_pile := !profondeur_pile - n * taille_du_mot;;
```

Les fonctions `réserve_pile` et `libère_pile` émettent du code machine qui modifie le registre pointeur de pile et modifient `profondeur_pile` pour que le code émis pour les accès aux variables relativement à ce registre reste correct.

Le code est émis sur la sortie standard à l'aide de la fonction `printf` du module de bibliothèque `printf`. Tout comme la fonction `sprintf` que nous avons rencontrée dans la section 11.2, `printf` prend en argument une chaîne de format et un certain nombre d'entiers ou de chaînes de caractères, et remplace dans le format les séquences de la forme `%` plus une lettre par le prochain argument. Au contraire de `sprintf`, qui renvoie le résultat du formatage sous forme de chaîne, `printf` l'affiche directement sur la sortie standard. Par exemple,

```
printf "add r %d, %s, r %d" 1 "étiquette" 2
```

affiche

```
add r 1, étiquette, r 2
```

La lettre suivant `%` indique le type de l'argument à afficher et le format d'affichage à utiliser. Parmi les nombreux formats existants, nous n'en utiliserons que deux : `%d`, qui affiche un entier en décimal, et `%s`, qui affiche une chaîne de caractères.

Fonctions d'analyse des expressions

Nous passons maintenant à des fonctions auxiliaires qui calculent certaines propriétés des expressions. La première calcule le type d'une expression de tableau, c'est-à-dire les bornes inférieures et supérieures du tableau, ainsi que le type des éléments ; cette dernière information détermine la taille des éléments lors d'un accès au tableau.

Fichier `compil.ml`

```
let rec type_de_tableau env = function
  | Variable nom ->
      begin match cherche_variable nom env with
        { typ = Array(inf, sup, ty) } -> (inf, sup, ty)
      end
  | Accès_tableau(arg1, arg2) ->
      match type_de_tableau env arg1 with
        (_, _, Array(inf, sup, ty)) -> (inf, sup, ty);;
```

Comme les tableaux ne peuvent être renvoyés comme résultat d'un appel de fonction, une expression bien typée de type `Array` ne peut être qu'une variable ou un accès dans un tableau multidimensionnel ; ces deux cas sont donc les seuls à considérer dans `type_de_tableau`.

La deuxième fonction auxiliaire détermine si l'évaluation d'une expression peut « interférer » avec l'évaluation d'une autre. Deux expressions n'interfèrent pas si leurs

résultats et les effets qu'elles effectuent ne dépendent pas de l'ordre dans lequel on les évalue. Une expression qui contient un appel de fonction est susceptible d'interférer avec une autre expression : la fonction peut écrire quelque chose à l'écran, ou modifier une variable globale. De même, une expression qui fait intervenir la valeur d'une variable globale peut interférer avec une autre expression qui va modifier cette variable globale à l'occasion d'un appel de fonction. On se convainc facilement qu'une expression ne contenant ni variables globales, ni appels de fonctions n'interfère avec aucune autre expression. (Pour rester simple, on suppose ici que tout appel de fonction est susceptible de faire des effets. Pour faire les choses plus finement, il faudrait analyser aussi le corps de la fonction appelée, pour voir s'il peut faire des effets en s'exécutant.)

Fichier `compil.ml`

```
let rec sans_interférences env = function
  | Constante c -> true
  | Variable nom ->
      let var = cherche_variable nom env in
      begin match var.emplacement with
      | Global_indirect _ | Global_direct _ -> false
      | Local_indirect _  | Local_direct _  -> true
      end
  | Application(fonc, args) -> false
  | Op_unaire(op, arg) ->
      sans_interférences env arg
  | Op_binaire(op, arg1, arg2) ->
      sans_interférences env arg1 && sans_interférences env arg2
  | Accès_tableau(arg1, arg2)  ->
      sans_interférences env arg1 && sans_interférences env arg2;;
```

La dernière fonction auxiliaire détermine les « besoins en registres » d'une expression : de combien de registres libres l'évaluation de l'expression a besoin pour stocker ses résultats intermédiaires. Cette information est précieuse pour déterminer quand il est nécessaire de sauvegarder des résultats intermédiaires sur la pile.

La stratégie que nous allons employer est la suivante : avant d'évaluer une expression, on regarde si ses besoins en registres sont supérieurs au nombre de registres inutilisés (c'est-à-dire ne contenant pas déjà un résultat intermédiaire) ; si oui, on libère certains des registres utilisés, en stockant leur valeur dans la pile avant l'évaluation de l'expression, puis en les rechargeant depuis la pile une fois l'expression évaluée.

Fichier `compil.ml`

```
let dernier_registre = 24;;

let rec besoins env = function
  | Constante c -> 0
  | Variable nom -> 0
  | Application(fonc, args) -> dernier_registre
  | Op_unaire(op, arg) -> besoins env arg
  | Op_binaire(op, arg1, arg2) -> besoins_op_binaire env arg1 arg2
  | Accès_tableau(arg1, arg2)  -> besoins_op_binaire env arg1 arg2

and besoins_op_binaire env arg1 arg2 =
```

```
let b1 = besoins env arg1 and b2 = besoins env arg2 in
if b1 < b2
 && (sans_interférences env arg1 || sans_interférences env arg2)
then max b2 (b1 + 1)
else max b1 (b2 + 1);;
```

L'évaluation des expressions peut utiliser tous les registres dont les numéros sont compris entre 1 et `dernier_registre` pour stocker des résultats intermédiaires. Les registres au-dessus de `dernier_registre` sont réservés pour d'autres usages (temporaires, pointeur de pile).

La constante `dernier_registre` est le nombre de registres du processeur que nous pouvons utiliser pour l'évaluation des expressions. On suppose que l'application d'une fonction ne préserve aucun de ces registres : la fonction appelée peut les utiliser librement, pour évaluer des expressions arbitrairement compliquées, sans obligation de sauvegarder leur contenu. Une application de fonction a donc besoin de tous les registres.

Pour une opération unaire comme `not` e, si on peut évaluer e avec n registres libres, on peut aussi évaluer `not` e avec n registres libres : il suffit d'évaluer e dans un certain registre r, puis de faire l'instruction `seq r 0`$, r, r$ qui ne nécessite pas de registre temporaire supplémentaire.

Le cas des opérations binaires est plus subtil. Pour évaluer $e_1 + e_2$, par exemple, on peut commencer par évaluer e_1, puis e_2, puis faire `add` sur les deux résultats. Mais il faut que la valeur de e_1 ne soit pas détruite pendant l'évaluation de e_2. Pour ce faire, on a besoin d'un registre libre supplémentaire pendant l'évaluation de e_2, registre dans lequel on conservera la valeur de e_1. Les besoins de l'expression $e_1 + e_2$ ainsi compilée sont donc le plus grand de `besoins` e_1 et de $1 +$ `besoins` e_2.

Si les expressions e_1 et e_2 sont susceptibles d'interférer, nous sommes obligés, pour respecter la sémantique de mini-Pascal, d'évaluer e_1 d'abord, puis e_2. En revanche, si l'une des expressions est garantie sans interférences, nous pouvons évaluer e_2 avant e_1 sans changer le comportement du programme. Si on évalue d'abord e_2 puis e_1, le nombre de registres utilisés est le plus grand de `besoins` e_2 et de $1 +$ `besoins` e_1. On choisit donc celle des deux stratégies qui utilise le moins de registres. On montre facilement que la stratégie « e_2 d'abord » utilise moins de registres que la stratégie « e_1 d'abord » si et seulement si `besoins` e_2 est plus grand que `besoins` e_1. La stratégie correspond donc à évaluer en premier celle des deux sous-expressions qui a les plus grands besoins en registres. (Cette méthode est connue dans la littérature sous le nom d'algorithme d'Ershov.)

Compilation d'une expression

Nous pouvons maintenant attaquer la fonction de compilation d'une expression. Cette fonction prend en arguments un environnement, une expression et un registre de destination et affiche sur la sortie standard le code machine qui calcule la valeur de cette expression et met le résultat dans le registre demandé. La plus grande partie de la fonction `compile_expr` se passe de commentaire. Nous utilisons toute la puissance du filtrage pour reconnaître certains cas particuliers qui correspondent directement à

des modes d'adressage de la machine, comme par exemple l'addition d'une constante ou le chargement avec un décalage constant.

Fichier `compil.ml`

```
let instr_pour_op = function
  | "+"   -> "add"     | "-"   -> "sub"
  | "*"   -> "mult"    | "/"   -> "div"
  | "="   -> "seq"     | "<>"  -> "sne"
  | "<"   -> "slt"     | ">"   -> "sgt"
  | "<="  -> "sle"     | ">="  -> "sge"
  | "and" -> "and"     | "or"  -> "or";;

let rec compile_expr env expr reg =
  match expr with
  | Constante cst ->
      printf "add r 0, %d, r %d\n" (val_const cst) reg
  | Variable nom ->
      let var = cherche_variable nom env in
      begin match var.emplacement with
      | Global_indirect n ->
          printf "load r 0, %d, r %d  # %s \n" n reg nom
      | Global_direct n ->
          printf "add r 0, %d, r %d  # %s \n" n reg nom
      | Local_indirect n ->
          printf "load sp, %d, r %d  # %s \n"
                 (!profondeur_pile - n) reg nom
      | Local_direct n ->
          printf "add sp, %d, r %d  # %s \n"
                 (!profondeur_pile - n) reg nom
      end
  | Application(fonc, arguments) ->
      let nbr_args = list_length arguments in
      réserve_pile nbr_args;
      let position = ref 0 in
      do_list (function arg ->
                 compile_expr env arg 1;
                 printf "store sp, %d, r 1\n" !position;
                 position := !position + taille_du_mot)
               arguments;
      printf "jmp F%s, ra\n" fonc;
      libère_pile nbr_args;
      if reg <> 1 then printf "add r 1, r 0, r %d\n" reg
  | Op_unaire(op, arg) ->
      compile_expr env arg reg;
      begin match op with
      | "-"   -> printf "sub r 0, r %d, r %d\n" reg reg
      | "not" -> printf "seq r 0, r %d, r %d\n" reg reg
      end
  | Op_binaire(op, arg1, Constante cst2) ->
      compile_expr env arg1 reg;
      printf "%s r %d, %d, r %d\n"
             (instr_pour_op op) reg (val_const cst2) reg
```

```
  | Op_binaire(("+" | "*" | "=" | "<>" | "and" | "or") as op,
              Constante cst1, arg2) ->
      compile_expr env arg2 reg;
      printf "%s r %d, %d, r %d\n"
             (instr_pour_op op) reg (val_const cst1) reg
  | Op_binaire(op, arg1, arg2) ->
      let (reg1, reg2) = compile_arguments env arg1 arg2 reg in
      printf "%s r %d, r %d, r %d\n" (instr_pour_op op) reg1 reg2 reg
  | Accès_tableau(arg1, Constante cst) ->
      let (inf, sup, type_éléments) = type_de_tableau env arg1 in
      compile_expr env arg1 reg;
      begin match type_éléments with
      | Integer | Boolean ->
          printf "load r %d, %d, r %d\n" reg
                 ((val_const cst - inf) * taille_du_mot) reg
      | Array(_, _, _) ->
          let taille = taille_du_type type_éléments in
          printf "add r %d, %d, r %d\n"
                 reg ((val_const cst - inf) * taille) reg
      end
  | Accès_tableau(arg1, arg2) ->
      let (inf, sup, type_éléments) = type_de_tableau env arg1 in
      let (reg1, reg2) = compile_arguments env arg1 arg2 reg in
      if inf <> 0 then printf "sub r %d, %d, r %d\n" reg2 inf reg2;
      begin match type_éléments with
      | Integer | Boolean ->
          printf "mult r %d, %d, r %d\n" reg2 taille_du_mot reg2;
          printf "load r %d, r %d, r %d\n" reg1 reg2 reg
      | Array(_, _, typ) ->
          let taille = taille_du_type type_éléments in
          printf "mult r %d, %d, r %d\n" reg2 taille reg2;
          printf "add r %d, r %d, r %d\n" reg1 reg2 reg
      end

and compile_arguments env arg1 arg2 reg_libre =
  let b1 = besoins env arg1 and b2 = besoins env arg2 in
  if b1 < b2
  && (sans_interférences env arg1 || sans_interférences env arg2)
  then begin
    let (reg2, reg1) = compile_arguments env arg2 arg1 reg_libre in
    (reg1, reg2)
  end else begin
    compile_expr env arg1 reg_libre;
    if b2 < dernier_registre - reg_libre then begin
      compile_expr env arg2 (reg_libre + 1);
      (reg_libre, reg_libre + 1)
    end else begin
      réserve_pile 1;
      printf "store sp, 0, r %d\n" reg_libre;
      compile_expr env arg2 reg_libre;
      printf "load sp, 0, r 29\n";
```

```
      libère_pile 1;
      (29, reg_libre)
    end
  end;;
```

La fonction `compile_arguments` implémente la stratégie d'évaluation présentée plus haut. Elle évalue deux expressions en séquence et renvoie les numéros de deux registres contenant leur valeur. Le dernier argument, `reg_libre`, est le numéro du premier registre libre. Tous les registres inférieurs à `reg_libre` seront préservés ; tous les autres peuvent être utilisés librement.

Si on peut permuter sans risques les deux évaluations et si cela peut réduire le nombre de registres utilisés (c'est-à-dire si `arg2` a de plus grands besoins que `arg1`), la fonction `compile_arguments` se rappelle après avoir permuté `arg2` et `arg1`.

Sinon, elle commence par émettre le code évaluant `arg1` dans `reg_libre`. Ensuite, `reg_libre` n'est plus libre (il contient la valeur de `arg1`). Donc, de deux choses l'une : ou bien on peut évaluer `arg2` avec les registres restants, auquel cas la fonction émet le code qui évalue `arg2` avec `reg_libre + 1` comme registre de destination ; ou bien on n'a plus assez de registres libres, auquel cas la fonction sauvegarde la valeur de `arg1` sur la pile (ce qui libère le registre `reg_libre`), émet le code qui évalue `arg2` dans `reg_libre` et recharge la valeur de `arg1` dans un registre temporaire (le registre 29).

Compilation des instructions

On passe maintenant à la compilation d'une instruction. La plupart des cas sont similaires à ceux de la compilation d'une expression.

Fichier `compil.ml`

```
let compteur_d'étiquettes = ref 0;;

let nouvelle_étiq () =
  incr compteur_d'étiquettes; !compteur_d'étiquettes;;

let rec compile_instr env = function
  | Affectation_var(nom_var,
                    Constante(Entière 0 | Booléenne false)) ->
      affecte_var env nom_var 0
  | Affectation_var(nom_var, expr) ->
      compile_expr env expr 1;
      affecte_var env nom_var 1
  | Affectation_tableau(expr1, Constante cst2, expr3) ->
      let (inf, sup, type_éléments) = type_de_tableau env expr1 in
      let (reg3, reg1) = compile_arguments env expr3 expr1 1 in
      printf "store r %d, %d, r %d\n"
             reg1 ((val_const cst2 - inf) * taille_du_mot) reg3
  | Affectation_tableau(expr1, expr2, expr3) ->
      let (inf, sup, type_éléments) = type_de_tableau env expr1 in
      compile_expr env expr3 1;
      let (reg1, reg2) = compile_arguments env expr1 expr2 2 in
      if inf <> 0 then printf "sub r %d, %d, r %d\n" reg2 inf reg2;
      printf "mult r %d, %d, r %d\n" reg2 taille_du_mot reg2;
```

```
    printf "store r %d, r %d, r %d\n" reg1 reg2 1
| Appel(proc, arguments) ->
    let nbr_args = list_length arguments in
    réserve_pile nbr_args;
    let position = ref 0 in
    do_list (function arg ->
              compile_expr env arg 1;
              printf "store sp, %d, r 1\n" !position;
              position := !position + taille_du_mot)
            arguments;
    printf "jmp P%s, ra\n" proc;
    libère_pile nbr_args
| If(condition, branche_oui, Bloc []) ->
    let étiq_fin = nouvelle_étiq () in
    compile_expr env condition 1;
    printf "braz r 1, L%d\n" étiq_fin;
    compile_instr env branche_oui;
    printf "L%d:\n" étiq_fin
| If(condition, Bloc [], branche_non) ->
    let étiq_fin = nouvelle_étiq () in
    compile_expr env condition 1;
    printf "branz r 1, L%d\n" étiq_fin;
    compile_instr env branche_non;
    printf "L%d:\n" étiq_fin
| If(Op_unaire("not", condition), branche_oui, branche_non) ->
    compile_instr env (If(condition, branche_non, branche_oui))
| If(condition, branche_oui, branche_non) ->
    let étiq_non = nouvelle_étiq ()
    and étiq_fin = nouvelle_étiq () in
    compile_expr env condition 1;
    printf "braz r 1, L%d\n" étiq_non;
    compile_instr env branche_oui;
    printf "braz r 0, L%d\n" étiq_fin;
    printf "L%d:\n" étiq_non;
    compile_instr env branche_non;
    printf "L%d:\n" étiq_fin
| While(condition, corps) ->
    let étiq_corps = nouvelle_étiq ()
    and étiq_test = nouvelle_étiq () in
    printf "braz r 0, L%d\n" étiq_test;
    printf "L%d:\n" étiq_corps;
    compile_instr env corps;
    printf "L%d:\n" étiq_test;
    compile_expr env condition 1;
    printf "branz r 1, L%d\n" étiq_corps
| Write expr ->
    compile_expr env expr 1;
    printf "write\n"
| Read nom_var ->
    printf "read\n";
    affecte_var env nom_var 1
```

```
  | Bloc liste_instr ->
      do_list (compile_instr env) liste_instr

and affecte_var env nom reg =
  let var = cherche_variable nom env in
  match var.emplacement with
  | Global_indirect n ->
      printf "store r 0, %d, r %d  # %s \n" n reg nom
  | Local_indirect n ->
      printf "store sp, %d, r %d  # %s \n"
             (!profondeur_pile - n) reg nom;;
```

Pour l'instruction `if` e `then` i_1 `else` i_2, le code produit a la forme suivante :

```
        code pour évaluer e dans le registre r 1
        braz r 1, Ln (branche si r 1 est false)
        code pour exécuter i1
        braz r 0, Lm (branche toujours)
Ln:     code pour exécuter i2
Lm:     suite du programme
```

Les étiquettes `L`n et `L`m sont de nouvelles étiquettes produites par la fonction `nouvelle_étiq`. Dans le cas où i_2 est l'instruction vide (cas d'un `if` sans partie `else`), on supprime le branchement à `L`m, qui ne sert à rien dans ce cas.

On procède de même pour l'instruction `while` e `do` i . Le test est placé à la fin du corps de la boucle, pour ne faire qu'un saut par tour de boucle au lieu de deux :

```
        braz r 0, Lm (branche toujours)
Ln:     code pour exécuter i
Lm:     code pour évaluer e dans le registre r 1
        branz r 1, Ln (branche si r 1 est true)
        suite du programme
```

Compilation des fonctions et des procédures

La compilation d'une fonction ou d'une procédure se fait en deux parties : il faut d'une part attribuer des emplacements aux paramètres et aux variables locales et construire l'environnement correspondant ; d'autre part, il faut émettre le code qui construit le bloc d'activation sur la pile. La seule subtilité est dans l'attribution des emplacements : une variable locale de type tableau a un emplacement de type `Local_direct`, puisque le tableau est alloué à plat dans la pile ; en revanche, un paramètre de type tableau a un emplacement de type `Local_indirect`, puisque c'est un pointeur vers le tableau passé en argument qui est empilé (passage par référence) et non pas le tableau lui-même (passage par valeur).

Fichier `compil.ml`

```
let alloue_variable_locale (nom, typ) env =
  profondeur_pile := !profondeur_pile + taille_du_type typ;
  let emplacement =
    match typ with
    | Integer | Boolean ->
```

```
        Local_indirect(!profondeur_pile)
    | Array(_, _, _) ->
        Local_direct(!profondeur_pile) in
  ajoute_variable nom {typ=typ; emplacement=emplacement} env;;

let alloue_paramètres liste_des_paramètres environnement =
  let prof = ref 0 in
  let env = ref environnement in
  do_list
   (function (nom,typ) ->
      env := ajoute_variable nom
              {typ=typ; emplacement = Local_indirect !prof}
              !env;
      prof := !prof - taille_du_mot)
    liste_des_paramètres;
  !env;;

let compile_procédure env (nom, décl) =
  let env1 =
    alloue_paramètres décl.proc_paramètres env in
  profondeur_pile := taille_du_mot;
  let env2 =
    list_it alloue_variable_locale décl.proc_variables env1 in
  printf "P%s:\n" nom;
  printf "sub sp, %d, sp\n" !profondeur_pile;
  printf "store sp, %d, ra\n" (!profondeur_pile - taille_du_mot);
  compile_instr env2 décl.proc_corps;
  printf "load sp, %d, ra\n" (!profondeur_pile - taille_du_mot);
  printf "add sp, %d, sp\n" !profondeur_pile;
  printf "jmp ra, r 0\n";;

let compile_fonction env (nom, décl) =
  let env1 =
    alloue_paramètres décl.fonc_paramètres env in
  profondeur_pile := taille_du_mot;
  let env2 =
    list_it alloue_variable_locale décl.fonc_variables env1 in
  let env3 =
    alloue_variable_locale (nom, décl.fonc_type_résultat) env2 in
  printf "F%s:\n" nom;
  printf "sub sp, %d, sp\n" !profondeur_pile;
  printf "store sp, %d, ra\n" (!profondeur_pile - taille_du_mot);
  compile_instr env3 décl.fonc_corps;
  printf "load sp, 0, r 1\n";
  printf "load sp, %d, ra\n" (!profondeur_pile - taille_du_mot);
  printf "add sp, %d, sp\n" !profondeur_pile;
  printf "jmp ra, r 0\n";;
```

Compilation d'un programme

Tout est prêt pour compiler un programme complet. Nous commençons par attribuer des adresses aux variables globales, obtenant ainsi l'environnement global de compilation, puis compilons successivement le corps du programme, les procédures et les fonctions dans cet environnement.

Fichier `compil.ml`

```
let adresse_donnée = ref 0;;

let alloue_variable_globale (nom, typ) env =
  let emplacement =
    match typ with
    | Integer | Boolean -> Global_indirect(!adresse_donnée)
    | Array(_, _, _)    -> Global_direct(!adresse_donnée) in
  adresse_donnée := !adresse_donnée + taille_du_type typ;
  ajoute_variable nom {typ=typ; emplacement=emplacement} env;;

let compile_programme prog =
  adresse_donnée := 0;
  let env_global =
    list_it alloue_variable_globale prog.prog_variables
            (environnement_initial prog.prog_procédures
                                   prog.prog_fonctions) in
  compile_instr env_global prog.prog_corps;
  printf "stop\n";
  do_list (compile_procédure env_global) prog.prog_procédures;
  do_list (compile_fonction env_global) prog.prog_fonctions;;
```

Le compilateur complet

Pour terminer, voici le programme principal qui combine l'analyseur syntaxique, le vérificateur de types et le compilateur.

Fichier `cpascal.ml`

```
#open "syntaxe";;

let compile_fichier nom =
  try
    let canal = open_in sys__command_line.(1) in
    try
      let prog = lire_programme (stream_of_channel canal) in
      close_in canal;
      typage__type_programme prog;
      compil__compile_programme prog
    with Parse_error | Parse_failure ->
           prerr_string "Erreur de syntaxe aux alentours \
                         du caractère numéro ";
           prerr_int (pos_in canal);
           prerr_endline ""
       | typage__Erreur_typage err ->
           typage__affiche_erreur err
```

```
  with sys__Sys_error message ->
        prerr_string "Erreur du système: "; prerr_endline message;;

if sys__interactive then () else
  begin compile_fichier sys__command_line.(1); exit 0 end;;
```

Mise en pratique

Pour compiler le tout :

```
$ camlc -c lexuniv.mli
$ camlc -c lexuniv.ml
$ camlc -c syntaxe.mli
$ camlc -c syntaxe.ml
$ camlc -c envir.mli
$ camlc -c envir.ml
$ camlc -c typage.mli
$ camlc -c typage.ml
$ camlc -c compil.mli
$ camlc -c compil.ml
$ camlc -c cpascal.ml
$ camlc -o cpascal lexuniv.zo syntaxe.zo envir.zo \
                   typage.zo compil.zo cpascal.zo
```

Lançons le compilateur sur le fichier `fib1.pas` donné en exemple page 277, par `camlrun cpascal fib1.pas` depuis l'interprète de commandes, ou par `cpascal__compile__fichier "fib1.pas"` depuis le système interactif. Nous obtenons le code suivant (la présentation a été légèrement modifiée pour être plus lisible).

```
      read
      store r 0, 0, r 1  # n
      sub sp, 4, sp
      load r 0, 0, r 1  # n
      store sp, 0, r 1
      jmp Ffib, ra
      add sp, 4, sp
      write
      stop
Ffib: sub sp, 8, sp
      store sp, 4, ra
      load sp, 8, r 1  # n
      slt r 1, 2, r 1
      braz r 1, L1
      add r 0, 1, r 1
      store sp, 0, r 1  # fib
      braz r 0, L2
L1:   sub sp, 4, sp
      load sp, 12, r 1  # n
      sub r 1, 1, r 1
      store sp, 0, r 1
      jmp Ffib, ra
      add sp, 4, sp
      sub sp, 4, sp
      store sp, 0, r 1
      sub sp, 4, sp
      load sp, 16, r 1  # n
      sub r 1, 2, r 1
      store sp, 0, r 1
      jmp Ffib, ra
      add sp, 4, sp
      load sp, 0, r 29
      add sp, 4, sp
      add r 29, r 1, r 1
      store sp, 0, r 1  # fib
L2:   load sp, 0, r 1
      load sp, 4, ra
      add sp, 8, sp
      jmp ra, r 0
```

Quoique loin d'être optimal, ce code n'est pas de trop mauvaise facture. L'inefficacité la plus grossière est la séquence `add sp, 4, sp; sub sp, 4, sp` au milieu de la

fonction `fib`, qui pourrait avantageusement être supprimée. De manière plus générale, il vaudrait mieux calculer à l'avance le nombre d'emplacements de pile nécessaires pour les temporaires et les allouer une fois pour toutes au début de la fonction, plutôt que d'incrémenter et de décrémenter le pointeur de pile à chaque fois qu'on a besoin d'un temporaire.

15.4 Pour aller plus loin

Le compilateur Pascal présenté dans ce chapitre se prête à de nombreuses extensions. Une première direction est d'enrichir le langage : pointeurs, nombres flottants, fonctions et procédures locales, ... En particulier, les fonctions et procédures locales posent des problèmes de compilation intéressants. Lorsque le langage interdit aux fonctions locales d'être renvoyées en résultats par d'autres fonctions, comme c'est le cas en Pascal, un chaînage des blocs d'activation dans la pile suffit. En revanche, dans les langages fonctionnels comme Caml où les fonctions sont des valeurs « de première classe », une pile ne suffit plus pour représenter les environnements d'évaluation des fonctions et il est nécessaire de modéliser les fonctions par des structures allouées dynamiquement : les fermetures (*closures*, en anglais). Nous reviendrons sur cette notion au chapitre 17.

Une deuxième direction est d'améliorer la qualité du code produit par le compilateur. En particulier, il faudrait essayer de stocker les variables locales autant que possible dans les registres et non pas dans la pile. Détecter les variables locales qui se prêtent à ce traitement et leur attribuer des registres de manière cohérente est un problème difficile.

Une troisième direction est d'améliorer la structure interne du compilateur. Nous avons vu que, pour afficher correctement les erreurs de typage, il faudrait annoter les nœuds de l'arbre de syntaxe abstraite par des numéros de ligne et des positions dans le code source. D'autres informations devraient elles aussi figurer en annotations sur l'arbre de syntaxe abstraite, comme le type attribué à chaque nœud par la phase de typage, ainsi que les besoins en registres de chaque expression. Cela éviterait de recalculer de nombreuses fois ces informations lors de la compilation.

Bibliographie

Parmi les nombreux ouvrages publiés sur les compilateurs, nous recommandons *Compilateurs : principes, techniques et outils*, de Aho, Sethi et Ullman (InterÉditions) et *Modern compiler implementation in ML*, d'Appel (Cambridge University Press).

16
Recherche de motifs dans un texte

Un programme pour les étoiles.

RECHERCHER LES OCCURRENCES d'un motif dans un texte est une opération cruciale dans de nombreux outils informatiques : traitement de texte, éditeurs, navigateurs Web, etc. Nous implémentons ici un outil qui affiche toutes les lignes d'un fichier contenant un motif donné, dans le style de la commande `grep` d'Unix. Pour ce faire, nous introduisons les notions d'automates et d'expressions rationnelles, qui sont un des fondements de l'informatique. Nous montrons comment manipuler en Caml des graphes et plus généralement des structures de données qui contiennent des cycles.

16.1 Les motifs

Dans le cas le plus simple, le motif que l'on recherche dans un fichier est une suite de caractères précise. Par exemple, en Unix, la commande `grep xop dictionnaire` affiche toutes les lignes du fichier `dictionnaire` qui contiennent la chaîne `xop`. Pour plus de souplesse dans la recherche, on autorise des « jokers » dans la chaîne à chercher. Ainsi, `grep c.r dictionnaire` affiche toutes les lignes contenant un `c` suivi d'une lettre quelconque puis d'un `r` ; de même `grep c.*r` affiche toutes les lignes contenant un `c` puis un `r`, séparés par un nombre quelconques de lettres.

Nous utiliserons une famille encore plus générale de motifs connus sous le nom d'expressions rationnelles (en anglais *regular expressions*). Une expression rationnelle est ou bien :

- un caractère c
- l'expression vide, notée ε
- une alternative $e_1 \mid e_2$, où e_1 et e_2 sont elles-mêmes deux expressions rationnelles
- une séquence $e_1 e_2$, où e_1 et e_2 sont elles-mêmes deux expressions rationnelles
- une répétition $e*$, où e est une expression rationnelle.

Pour chaque expression rationnelle, on définit les chaînes de caractères reconnues par cette expression rationnelle.

- L'expression c, où c est un caractère, reconnaît la chaîne à un seul caractère c, et rien d'autre.
- L'expression ε reconnaît la chaîne vide et rien d'autre.
- L'expression $e_1 \mid e_2$ reconnaît les chaînes qui sont reconnues par e_1 ou par e_2.
- L'expression $e_1 e_2$ reconnaît les chaînes composées d'une chaîne reconnue par e_1 suivie d'une chaîne reconnue par e_2.
- L'expression $e*$ reconnaît les chaînes composées de zéro, une ou plusieurs chaînes toutes reconnues par e.

Par exemple, l'expression `cal(i|y)(ph|f)e` reconnaît quatre orthographes envisageables pour le mot `calife`. De même, l'expression

$$(-|+|\varepsilon)\ (0|1|2|3|4|5|6|7|8|9)\ (0|1|2|3|4|5|6|7|8|9)*$$

reconnaît toutes les représentations décimales de nombres entiers, éventuellement précédées d'un signe.

16.2 Syntaxe abstraite et syntaxe concrète des motifs

Commençons par construire un module `expr` définissant la représentation interne (syntaxe abstraite) des expressions rationnelles, ainsi qu'une fonction pour lire une représentation textuelle (syntaxe concrète) d'une expression rationnelle. Voici l'interface de ce module :

Fichier `expr.mli`

```
type expr =
   | Epsilon
   | Caractères of char list
   | Alternative of expr * expr
   | Séquence of expr * expr
   | Répétition of expr;;

value lire : char stream -> expr;;
```

La syntaxe abstraite (le type concret `expr`) suit de très près la définition des expressions rationnelles donnée ci-dessus. La seule différence est que, pour des raisons d'efficacité, le constructeur `Caractères` prend en argument non pas un seul caractère, mais une liste de caractères. L'expression `Caractères`$[c_1; \ldots; c_n]$ représente l'alternative $c_1 \mid \ldots \mid c_n$.

La syntaxe concrète des expressions rationnelles ressemble à celle employée par `grep`. Elle introduit un certain nombre de constructions syntaxiques dérivées des constructions de base (alternative, séquence, répétition).

- Le point `.` représente n'importe quel caractère, c'est-à-dire l'alternative entre tous les caractères.

- Un caractère non spécial représente l'expression mono-caractère correspondante. La barre oblique inversée (*backslash*) sert d'échappement pour les caractères spéciaux : `\` suivi de n'importe quel autre caractère représente ce caractère. En particulier, `\\` est le caractère `\`.
- Les crochets `[ ...]` représentent des ensembles de caractères. Par exemple, `[aeiou]` représente `a|e|i|o|u`. Le tiret `-` dénote des intervalles : `[0-9]` représente tous les chiffres, `[A-Za-z]` toutes les lettres. On prend le complémentaire d'un ensemble en mettant un chapeau `^` juste après le crochet : `[^0-9]` représente tout caractère qui n'est pas un chiffre.
- Les combinaisons d'expressions rationnelles sont notées comme suit :

Combinaison	Notation	Exemple	Signification
Alternative	`|` infixe	`le|la|les`	`le`, ou `la`, ou `les`
Séquence	concaténation	`x[0-9]`	`x` puis un chiffre
Répétition	`*` postfixe	`[0-9]*`	zéro, un ou plusieurs chiffres
Répétition stricte	`+` postfixe	`[0-9]+`	un ou plusieurs chiffres
Option	`?` postfixe	`[+-]?`	un signe plus, un signe moins, ou rien

  L'alternative a la priorité la plus faible, puis la séquence, puis les répétitions. Pour passer outre ces priorités, on dispose des parenthèses `(...)`.
- Un signe chapeau `^` en début d'expression rationnelle signifie que le mot reconnu doit apparaître en début de ligne. Un signe dollar `$` en fin d'expression rationnelle signifie que le mot reconnu doit se trouver en fin de ligne. Par défaut, le mot reconnu se situe n'importe où dans la ligne.

Notation	Reconnaît ...	Codage
`^`*e*`$`	les lignes reconnues par *e*	*e*
`^`*e*	les lignes commençant par un mot reconnu par *e*	*e*`.*`
e`$`	les lignes finissant par un mot reconnu par *e*	`.*`*e*
e	les lignes contenant un mot reconnu par *e*	`.*`*e*`.*`

Voici maintenant l'implémentation du module `expr`, qui fournit l'analyseur pour la syntaxe qu'on vient de décrire.

Fichier `expr.ml`

```
let intervalle c1 c2 =
  let rec interv n1 n2 =
    if n1 > n2 then [] else char_of_int n1 :: interv (n1 + 1) n2 in
  interv (int_of_char c1) (int_of_char c2);;

let tous_car = intervalle '\000' '\255';;
```

La fonction `intervalle` construit la liste de tous les caractères entre les deux caractères donnés. Elle sert pour l'expansion des classes de caractères. La liste `tous_car` est la liste des 256 caractères du code ASCII. Elle sert pour l'expansion de la construction « . » en la classe de tous les caractères.

L'essentiel de l'analyse syntaxique est effectué par le groupe de fonctions mutuellement récursives ci-dessous. (La syntaxe d'entrée est si primitive que nous n'avons pas besoin d'une phase préalable d'analyse lexicale.) Le découpage en plusieurs fonctions intermédiaires assure en particulier que les priorités entre opérateurs sont respectées.

Fichier `expr.ml`

```
let rec lire_expr = function
  | [< lire_séq r1; (lire_alternative r1) r2 >] -> r2

and lire_alternative r1 = function
  | [< ''|'; lire_expr r2 >] -> Alternative(r1,r2)
  | [< >] -> r1

and lire_séq = function
  | [< lire_répét r1; (lire_fin_séq r1) r2 >] -> r2

and lire_fin_séq r1 = function
  | [< lire_séq r2 >] -> Séquence(r1,r2)
  | [< >] -> r1

and lire_répét = function
  | [< lire_simple r1; (lire_fin_répét r1) r2 >] -> r2

and lire_fin_répét r1 = function
  | [< ''* >] -> Répétition r1
  | [< ''+' >] -> Séquence(r1, Répétition r1)
  | [< ''?' >] -> Alternative(r1, Epsilon)
  | [< >] -> r1

and lire_simple = function
  | [< ''.' >] -> Caractères tous_car
  | [< ''['; lire_classe cl >] -> Caractères cl
  | [< ''('; lire_expr r; '')' >] -> r
  | [< ''\\'; 'c >] -> Caractères [c]
  | [< (stream_check
         (function c -> c <> '|' && c <> ')' && c <> '$')) c >] ->
      Caractères [c]

and lire_classe = function
  | [< ''^'; lire_ensemble cl >] -> subtract tous_car cl
  | [< lire_ensemble cl >] -> cl

and lire_ensemble = function
  | [< '']' >] -> []
  | [< lire_car c1; (lire_intervalle c1) c2 >] -> c2

and lire_intervalle c1 = function
```

```
  | [< ''-'; lire_car c2; lire_ensemble reste >] ->
       union (intervalle c1 c2) reste
  | [< lire_ensemble reste >] -> union [c1] reste

and lire_car = function
  | [< ''\\'; 'c >] -> c
  | [< 'c >] -> c;;
```

Nous avons dû introduire les fonctions intermédiaires `lire_alternative`, `lire_fin_séq`, etc., pour tenir compte du caractère entièrement déterministe du filtrage sur les flux. Pour `lire_expr` par exemple, il aurait été plus naturel d'écrire :

```
let rec lire_expr = function
  | [< lire_séq r1; ''|'; lire_expr r2 >] -> Alternative(r1,r2)
  | [< lire_séq r1 >] -> r1
```

Cette écriture ne donne pas le résultat attendu : si `lire_séq` reconnaît le début du flux, on s'engage de manière définitive dans le premier cas du filtrage. Si le prochain caractère du flux n'est pas une barre verticale, le système déclenche une erreur de syntaxe, mais ne se rabat pas sur le deuxième cas du filtrage.

Il faut donc appliquer aux motifs des fonctions d'analyse la technique connue sous le nom de « factorisation à gauche » : la fonction `lire_expr` commence par reconnaître le préfixe commun aux deux cas, c'est-à-dire `lire_séq`, puis appelle la fonction d'analyse auxiliaire `lire_alternative` pour tester la présence de la barre verticale. S'il y en a une, le premier cas de `lire_alternative` est sélectionné, et appelle récursivement `lire_expr` pour lire l'expression qui suit. Sinon, `lire_alternative` ne lit rien et `lire_expr` renvoie simplement l'expression lue par `lire_séq`.

Fichier `expr.ml`

```
let lire = function
| [< (function [< ''^' >] -> true | [< >] -> false) chapeau;
     lire_expr r;
     (function [< ''$' >] -> true | [< >] -> false) dollar >] ->
  let r1 = if dollar then r else
             Séquence(r, Répétition(Caractères tous_car)) in
  if chapeau then r1 else
    Séquence(Répétition(Caractères tous_car), r1);;
```

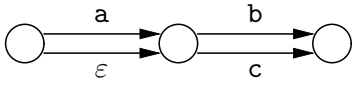
La fonction `lire`, point d'entrée du module, lit une expression rationnelle éventuellement précédée d'un caractère `^` ou suivie d'un caractère `$`. Remarquez que, depuis l'intérieur d'un motif de flux, il est possible d'appeler des fonctions d'analyse anonymes (non nommées), introduites par `function`.

16.3 Les automates

Pour programmer la commande `grep`, il faut savoir déterminer si une expression rationnelle reconnaît une chaîne de caractères. La traduction naïve de la définition des chaînes reconnues par une expression rationnelle mène à un algorithme par essais et erreurs qui est très inefficace dans les cas défavorables. Intuitivement, pour reconnaître

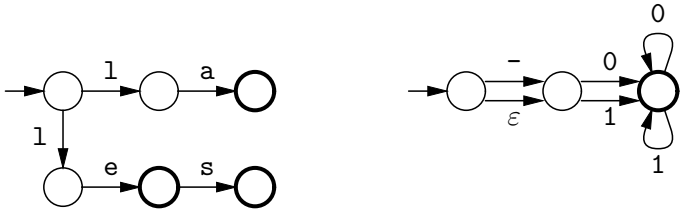
une expression de la forme `.*`e`.*`, il faut essayer de reconnaître e à toutes les positions possibles dans la chaîne. Si plusieurs expressions de ce type se suivent, comme dans `.*`e_1`.*`$e_2 \ldots e_n$`.*`, le nombre de combinaisons à essayer devient très grand.

La manière efficace de déterminer si une expression rationnelle reconnaît une chaîne de caractères est de transformer d'abord l'expression rationnelle en un *automate* qui reconnaît les mêmes mots, puis d'exécuter l'automate sur ladite chaîne de caractères. Intuitivement, un automate est une machine très simplifiée, spécialisée dans la reconnaissance des mots. Elle se compose d'un certain nombre d'*états* (représentés par des cercles) reliés entre eux par des *transitions* (représentées par des flèches). Les transitions sont étiquetées ou bien par une lettre, ou bien par le mot vide ε.



La règle du jeu est la suivante : à partir d'un état, on a le droit de passer dans un autre état soit parce qu'il existe une transition marquée ε de l'état de départ vers l'état d'arrivée, soit parce qu'il existe une transition marquée x (où x est le prochain caractère de la chaîne à reconnaître) de l'état de départ vers l'état d'arrivée. Par exemple, dans l'automate ci-dessus, on peut passer de l'état de gauche à l'état de droite si la chaîne à reconnaître commence par `ab`, `ac`, `b`, ou `c`.

Dans un automate, certains états sont marqués comme états *terminaux*. Un des états est marqué comme état *initial*. (Dans les dessins, l'état initial est signalé par une petite flèche entrante du côté gauche ; les états terminaux sont en trait plus épais.)



Le jeu consiste à essayer de trouver un chemin qui part de l'état initial et aboutit sur un état terminal, après avoir lu tous les caractères de la chaîne donnée en entrée. Si un tel chemin existe, on dit que l'automate reconnaît la chaîne. Par exemple, l'automate ci-dessus à gauche reconnaît les mots `le`, `la`, `les`, et rien d'autre. L'automate ci-dessus à droite reconnaît les nombres écrits en base deux, c'est-à-dire les mêmes mots que l'expression rationnelle `-?[01]+`.

16.4 Des expressions rationnelles aux automates

Expressions rationnelles et automates sont reliés de manière très étroite : à toute expression rationnelle correspond un automate qui reconnaît exactement les mêmes mots que l'expression de départ. Les automates peuvent donc être vus comme des formes compilées d'expressions rationnelles.

Nous allons maintenant programmer une fonction qui transforme une expression rationnelle en automate. Le module correspondant s'appelle `auto` ; voici son interface.

Fichier `auto.mli`

```
#open "expr";;

type état =
  { mutable transitions : (char * état) list;
    mutable epsilon_transitions : état list;
    mutable terminal : bool;
    numéro : int };;

value expr_vers_automate : expr -> état;;
```

Un état de l'automate est représenté par un enregistrement à quatre champs. Le champ `terminal` indique si l'état est terminal ou non. Les champs `transitions` et `epsilon_transitions` contiennent la liste des flèches sortant de l'état, avec pour chaque flèche l'état auquel elle mène. Le champ `numéro` sert à identifier les états de manière unique : deux états différents portent des numéros différents.

L'automate lui-même est représenté par son état initial. Les autres états de l'automate « pendent » sous l'état initial : ils sont accessibles en descendant dans les champs `transitions` et `epsilon_transitions`.

L'implémentation du module `auto` comporte deux parties : premièrement, quelques petites fonctions de manipulation des états ; deuxièmement, la fonction de compilation d'une expression rationnelle en automate.

Fichier `auto.ml`

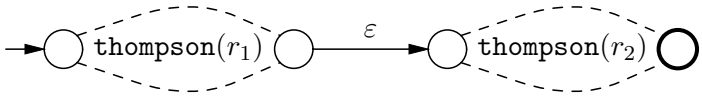
```
#open "expr";;

let compteur_d'états = ref 0;;

let nouvel_état () =
  incr compteur_d'états;
  { transitions = []; epsilon_transitions = [];
    terminal = false; numéro = !compteur_d'états };;

let ajoute_trans n1 c n2 =
  n1.transitions <- (c, n2) :: n1.transitions;;

let ajoute_eps_trans n1 n2 =
  n1.epsilon_transitions <- n2 :: n1.epsilon_transitions;;

type automate_de_thompson =
  { initial : état;
    final   : état };;

let rec thompson = function
  | Epsilon ->
      let e1 = nouvel_état () and e2 = nouvel_état () in
      ajoute_eps_trans e1 e2;
      {initial = e1; final = e2}
  | Caractères cl ->
      let e1 = nouvel_état () and e2 = nouvel_état () in
      do_list (function c -> ajoute_trans e1 c e2) cl;
```

```
      {initial = e1; final = e2}
  | Alternative(r1, r2) ->
      let t1 = thompson r1 and t2 = thompson r2 in
      let e1 = nouvel_état () and e2 = nouvel_état () in
      ajoute_eps_trans e1 t1.initial; ajoute_eps_trans e1 t2.initial;
      ajoute_eps_trans t1.final e2;   ajoute_eps_trans t2.final e2;
      {initial = e1; final = e2}
  | Séquence(r1, r2) ->
      let t1 = thompson r1 and t2 = thompson r2 in
      ajoute_eps_trans t1.final t2.initial;
      {initial = t1.initial; final = t2.final}
  | Répétition r ->
      let t = thompson r in
      let e1 = nouvel_état () and e2 = nouvel_état () in
      ajoute_eps_trans t.final t.initial;
      ajoute_eps_trans e1 t.initial;
      ajoute_eps_trans t.final e2;
      ajoute_eps_trans e1 e2;
      {initial = e1; final = e2};;

let expr_vers_automate r =
  let t = thompson r in t.final.terminal <- true; t.initial;;
```

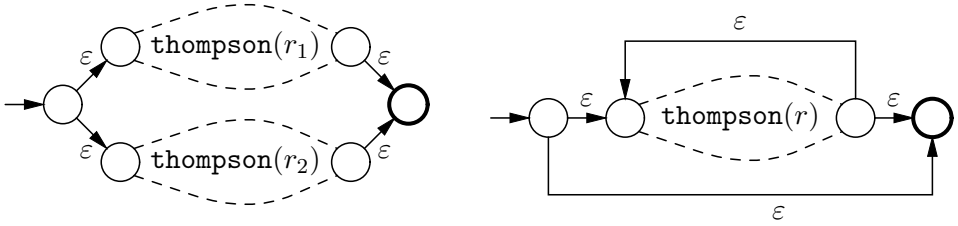
L'algorithme de transformation d'une expression rationnelle en automate employé ci-dessus est connu sous le nom de « construction de Thompson ». Les automates qu'il produit ont la particularité d'avoir un seul état terminal, qu'on appelle l'état final, par symétrie avec l'état initial. De plus, aucune transition ne sort de l'état final. On introduit le type `automate_de_thompson` pour garder en même temps un pointeur sur l'état initial et un pointeur sur l'état final de l'automate.

La construction de Thompson procède par récurrence sur la structure de l'expression rationnelle. Pour les deux cas de base, `Epsilon` et `Caractères`$[c_1; \ldots; c_n]$, on renvoie les deux automates suivants :

Clairement, l'automate de gauche reconnaît uniquement le mot vide, et l'automate de droite uniquement les chaînes mono-caractères $c_1, \ldots, c_n$. Pour le cas `Séquence`(r_1, r_2), on construit récursivement les automates de Thompson correspondant à r_1 et r_2, et on met une transition étiquetée ε de l'état final de l'automate pour r_1 vers l'état initial de l'automate pour r_2.



L'automate résultant reconnaît les concaténations d'un mot reconnu par `thompson` r_1 et d'un mot reconnu par `thompson` r_2. Il reconnaît donc bien les mêmes mots que l'expression rationnelle `Séquence`(r_1, r_2). En suivant un raisonnement semblable, on prend pour les cas `Alternative`(r_1, r_2) et `Répétition`(r) :



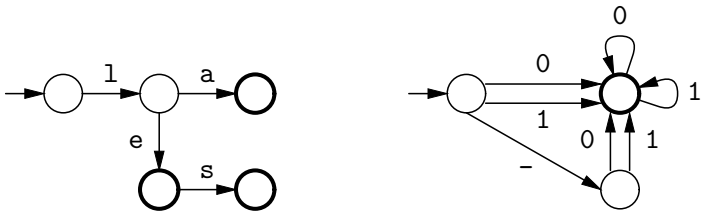
La fonction `expr_vers_automate` est alors très simple : on construit l'automate de Thompson associé à l'expression rationnelle, on marque son état final comme étant terminal, et on renvoie l'état initial.

16.5 Déterminisation de l'automate

Ayant obtenu un automate qui reconnaît les mêmes chaînes que l'expression rationnelle de départ, il nous reste à programmer une fonction qui teste si une chaîne est reconnue ou non par l'automate. Ce test n'est pas immédiat en général : puisque plusieurs transitions portant le même caractère peuvent sortir d'un même état, il faut quelquefois essayer plusieurs chemins qui épellent la chaîne à reconnaître. Cependant, il existe une classe d'automates, les automates déterministes, pour lesquels le problème est beaucoup plus simple. Un automate déterministe a les deux propriétés suivantes :

1. il ne possède pas de transition marquée ε
2. d'un même état il ne part jamais plusieurs transitions marquées par le même caractère.

Voici par exemple deux automates déterministes reconnaissant `le`, `la`, `les` pour celui de gauche, et les entiers en base deux pour celui de droite.



Pour tester si une chaîne est reconnue par un automate déterministe, il suffit de partir de l'état initial et de suivre à chaque état la transition (unique) marquée par le prochain caractère de la chaîne. Si on aboutit sur un état terminal, la chaîne est reconnue. Si on aboutit sur un état non terminal, ou si on reste bloqué en cours de route parce qu'un état n'a pas de transition sur le prochain caractère, alors la chaîne n'est pas reconnue.

La théorie des automates montre que pour tout automate il existe un automate déterministe qui reconnaît exactement les mêmes chaînes. Nous allons donc commencer par transformer l'automate précédemment construit en un automate déterministe, puis utiliser cet automate déterministe pour décider si une chaîne est reconnue ou pas. Voici l'interface du module `determ`, qui fournit ces deux fonctions.

Fichier `determ.mli`

```
type état =
  { mutable dtransitions : transition vect;
    dterminal : bool }
and transition =
  | Vers of état
  | Rejet;;

value déterminise : auto__état -> determ__état
  and reconnaît : determ__état -> string -> bool;;
```

Un état d'un automate déterministe est représenté par un enregistrement à deux champs : un booléen `dterminal`, indiquant si l'état est terminal ou non, et un tableau `dtransitions` à 256 cases, une par caractère du jeu ASCII. Le constructeur `Vers` indique la présence d'une transition vers l'état indiqué ; le constructeur `Rejet` indique l'absence de transition.

L'implémentation de la fonction `reconnaît` est très simple.

Fichier `determ.ml`

```
exception Échec;;

let reconnaît automate chaîne =
  let état_courant = ref automate in
  try
    for i = 0 to string_length chaîne - 1 do
    match !état_courant.dtransitions.(int_of_char chaîne.[i])
    with Rejet  -> raise Échec
       | Vers e -> état_courant := e
    done;
    !état_courant.dterminal
  with Échec -> false;;
```

Le reste du fichier `determ.ml` est consacré à la fonction de déterminisation d'un automate. L'algorithme utilisé est connu sous le nom de « construction des sous-ensembles » (*subset construction*). Les états de l'automate déterministe correspondent à des ensembles d'états de l'automate de départ : tous les états qu'on peut atteindre à partir de l'état initial en suivant une certaine chaîne de caractères.

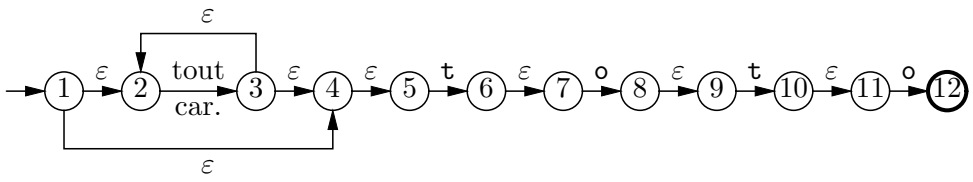
L'état initial de l'automate déterministe est l'ensemble des états qu'on peut atteindre en suivant la chaîne vide, c'est-à-dire l'état initial de l'automate de départ, plus tous les états qu'on peut atteindre à partir de l'état initial en suivant uniquement des epsilon-transitions (des transitions marquées ε).

L'état correspondant à l'ensemble d'états $\{e_1, \ldots, e_n\}$ est terminal si et seulement si un des états $e_1, \ldots, e_n$ est terminal.

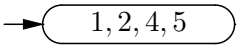
Pour voir où mène la transition sur un caractère c issue de l'ensemble d'états $\{e_1, \ldots, e_n\}$, on regarde toutes les transitions sur c issues des états e_1 à e_n dans l'automate initial. Soient $f_1, \ldots, f_m$ les états auxquels elles mènent. Soient $g_1, \ldots, g_p$ les états accessibles à partir de $f_1, \ldots, f_m$ en suivant uniquement des epsilon-transitions. On ajoute alors, dans l'automate déterministe produit, une transition sur c depuis l'état

$\{e_1, \ldots, e_n\}$ vers l'état $\{f_1, \ldots, f_m, g_1, \ldots, g_p\}$. On répète ce procédé jusqu'à ce qu'il soit impossible d'ajouter de nouvelles transitions.

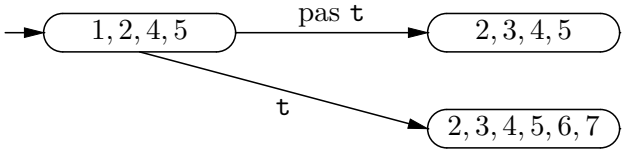
À titre d'exemple, nous allons déterminiser l'automate produit pour l'expression `.*toto` par la fonction `expr_vers_automate`.



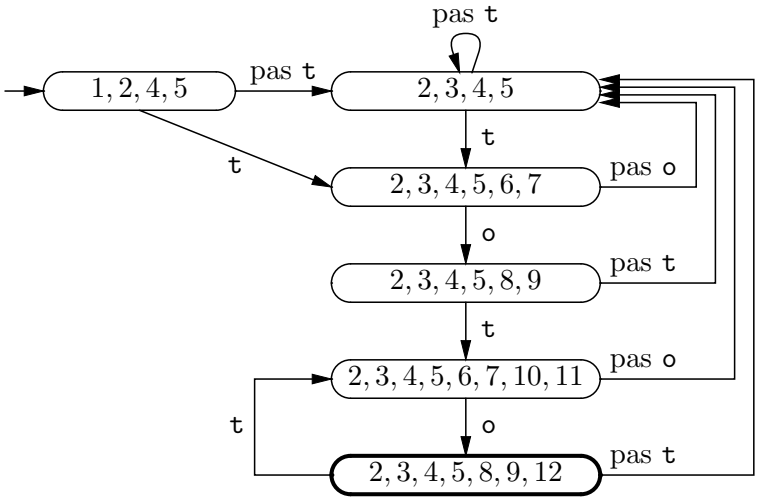
Les états sont numérotés de 1 à 12 pour être repérés plus facilement. À partir de l'état initial 1, on peut atteindre 2, 4 et 5 par epsilon-transitions. L'état initial de l'automate déterministe est donc $\{1, 2, 4, 5\}$. Il est non terminal.



Il y a deux types de transitions issues de cet état : la transition sur `t` et les transitions sur un caractère autre que `t`. Sur `t`, 2 mène à 3, 5 mène à 6, 2 et 4 ne mènent à rien. À partir de 3, on peut atteindre 2, 4, 5 par epsilon-transitions. À partir de 6, on peut atteindre 7 par epsilon-transition. On ajoute donc une transition sur `t` issue de $\{1, 2, 4, 5\}$ et menant à $\{2, 3, 4, 5, 6, 7\}$. De même, sur un caractère autre que `t`, 2 mène à 3 et 2, 4, 5 ne mènent à rien. A partir de 3, on peut atteindre 2, 4, 5 par epsilon-transitions. On ajoute donc des transitions sur tous les caractères sauf `t`, transitions issues de $\{1, 2, 4, 5\}$ et menant à $\{2, 3, 4, 5\}$.



En répétant ce raisonnement jusqu'à plus soif, on finit par obtenir l'automate déterministe suivant :



Seul l'état $\{2, 3, 4, 5, 8, 9, 12\}$ est terminal, puisque c'est le seul à contenir l'état terminal 12 de l'automate de départ.

Nous allons maintenant implémenter cet algorithme de déterminisation en Caml. La première chose à faire est de fournir une représentation des ensembles d'états, ainsi que les opérations de base sur ces ensembles.

Fichier `determ.ml`

```
#open "auto";;

type ensemble_d'états =
  { contenu  : ensent__t;
    éléments : auto__état list };;
let vide = { contenu = ensent__vide; éléments = [] };;
let est_vide ens =
  match ens.éléments with [] -> true | _ -> false;;
let appartient état ens =
  ensent__appartient état.numéro ens.contenu;;
let ajoute état ens =
  { contenu  = ensent__ajoute état.numéro ens.contenu;
    éléments = état :: ens.éléments };;
```

Un ensemble d'états est représenté par la liste des états appartenant à l'ensemble (champ `éléments`), et par un ensemble d'entiers (champ `contenu`) : les numéros des états appartenant à l'ensemble. On se donne un module `ensent` qui implémente le type `ensent__t` des ensembles d'entiers. Voici son interface ; on donne en annexe (section 16.7) une implémentation possible de ce module.

Fichier `ensent.mli`

```
type t;;
value vide : t
  and appartient : int -> t -> bool
  and ajoute : int -> t -> t;;
```

Cette représentation apparemment redondante des ensembles d'états est bien adaptée à l'utilisation qu'on en fait par la suite : le champ `éléments` permet d'itérer facilement sur tous les états d'un ensemble ; le champ `contenu` permet de tester efficacement l'appartenance et l'égalité entre ensembles. (La primitive d'égalité structurelle `=` n'est pas utilisable pour comparer des états, parce que la structure qui « pend » sous un état est susceptible de contenir des cycles, qui font boucler l'égalité.)

Viennent ensuite les deux opérations de base de l'algorithme de déterminisation : d'une part, ajouter à un état ou ensemble d'états tous les états qu'on peut atteindre par des epsilon-transitions (ce qu'on appelle prendre la « fermeture » d'un état ou ensemble d'états) ; d'autre part, déterminer les transitions possibles à partir d'un ensemble d'états.

Fichier `determ.ml`

```
let rec ajoute_fermeture état ferm =
  if appartient état ferm then ferm else
    list_it ajoute_fermeture
            état.epsilon_transitions (ajoute état ferm);;

let fermeture état = ajoute_fermeture état vide;;
```

```
let fermeture_ens ens = list_it ajoute_fermeture ens.éléments vide;;
```

On appelle fermeture d'un état l'ensemble des états accessibles depuis cet état en zéro, une ou plusieurs epsilon-transitions. À partir de l'état e, on atteint e et aucun autre état en zéro epsilon-transitions. En une ou plusieurs transitions, on passe d'abord par un état e' accessible à partir de e par une epsilon-transition, puis on atteint un des états appartenant à la fermeture de e'.

Cela suggère la définition suivante de la fonction `fermeture` : `fermeture`(e) est l'union du singleton $\{e\}$ et des ensembles `fermeture`(e') pour e' décrivant e.`epsilon_transitions`.

Cette définition est incorrecte à cause des cycles possibles : par exemple si e possède une epsilon-transition vers e' qui a une epsilon-transition vers e. Pour contourner cette difficulté, l'idée est de garder un ensemble des états qu'on sait d'ores et déjà appartenir à la fermeture. S'il se présente un état e qui n'est pas dans cet ensemble, on l'ajoute et on examine récursivement tous les états de e.`epsilon_transitions`. Si l'état e est déjà dans cet ensemble, il n'y a rien à faire. La fonction `ajoute_fermeture` implémente cet algorithme. Les fonctions `fermeture` et `fermeture_ens` sont de simples applications de `fermeture` obtenues en prenant l'ensemble vide comme ensemble d'états déjà vus.

Fichier `determ.ml`

```
let déplacements liste_états =
  let t = make_vect 256 vide in
  do_list
    (function état ->
      do_list
        (function (car, dest) ->
          let i = int_of_char car in t.(i) <- ajoute dest t.(i))
      état.transitions)
    liste_états;
  t;;
```

La fonction `déplacements` ci-dessus calcule toutes les transitions possibles à partir d'un ensemble d'états. Elle renvoie un tableau de 256 ensembles d'états, qui fait correspondre à chaque caractère l'ensemble des états atteints par transition sur ce caractère. Lorsque cet ensemble est vide, cela signifie qu'il n'y a pas de transition possible sur ce caractère.

Nous pouvons maintenant attaquer la fonction de déterminisation proprement dite. L'idée de départ est simple : pour calculer l'état de l'automate déterministe correspondant à un ensemble d'états e, on calcule `déplacements`(e), et pour chaque transition possible, on calcule la fermeture de l'ensemble d'états destination, puis on construit par un appel récursif l'état destination de la transition, qui est l'état correspondant à cette fermeture. Traduisant directement cette approche en Caml Light, on prendrait :

```
let rec déterminise ens =
  { dterminal = exists (function e -> e.terminal) ens.éléments;
    dtransitions = map_vect déterm_trans (déplacements ens) }
and déterm_trans dest =
  if est_vide dest
```

```
  then Rejet
  else Vers(déterminise(fermeture_ens dest));;
```

Cette approche est malheureusement inadaptée à la structure de graphe des automates : dans le cas d'une transition qui reboucle sur l'état dont elle est issue, la fonction `déterminise` va se rappeler à l'infini sur le même ensemble d'états.

Pour contourner cette difficulté, il faut séparer construction d'un nouvel état et calcul des transitions issues de cet état : les états sont créés initialement sans transitions sortantes, et on les complète ultérieurement en ajoutant les transitions convenables.

Fichier `determ.ml`

```
let déterminise état_initial =
  let états_connus = hashtbl__new 51
  and à_remplir = stack__new () in
  let traduire ens =
    try hashtbl__find états_connus ens.contenu
    with Not_found ->
      let nouvel_état =
        { dterminal = exists (function n -> n.terminal) ens.éléments;
          dtransitions = make_vect 256 Rejet } in
      hashtbl__add états_connus ens.contenu nouvel_état;
      stack__push (ens.éléments, nouvel_état) à_remplir;
      nouvel_état in
  let nouvel_état_initial =
    traduire (fermeture état_initial) in
  begin try
    while true do
      let (liste, nouvel_état) = stack__pop à_remplir in
      let dépl = déplacements liste in
      for i = 0 to 255 do
        if not est_vide dépl.(i) then
          nouvel_état.dtransitions.(i) <-
            Vers(traduire (fermeture_ens dépl.(i)))
      done
    done
  with stack__Empty -> ()
  end;
  nouvel_état_initial;;
```

Le cœur de la fonction de déterminisation est la fonction `traduire`, qui prend en argument un ensemble d'états de l'automate de départ et renvoie l'état correspondant de l'automate déterministe. Si l'ensemble d'états a déjà été rencontré, on trouve dans la table `états_connus` l'état associé. Sinon, on crée un nouvel état, on l'associe à l'ensemble d'états dans la table `états_connus`, et on le renvoie. Pour éviter le bouclage, on n'essaye pas de calculer immédiatement les transitions issues du nouvel état : ce dernier est créé sans aucune transition sortante. On se contente de le mettre dans la pile `à_remplir`, qui garde trace des états incomplets, dont il faudra déterminer les transitions plus tard.

Pour terminer la construction de l'automate, il faut dépiler les états incomplets, déterminer les transitions sortantes en appelant `déplacements` et `fermeture_ens`, et

obtenir les états de destination des transitions en appelant `traduire`. Les appels à `traduire` construisent parfois de nouveaux états ; il faut donc répéter ce processus jusqu'à ce que la pile `à_remplir` soit vide. (La terminaison est assurée parce que le nombre d'ensembles d'états possibles est fini : si l'automate initial a n états, il y a au plus 2^n ensembles d'états différents à considérer.)

On amorce le processus en appelant `traduire` sur la fermeture de l'état initial. On obtient ainsi l'état initial de l'automate déterministe. Comme ses transitions n'ont pas été calculées, la pile `à_remplir` contient cet état. Le premier tour de la boucle `while` complète cet état comme décrit ci-dessus. Les tours suivants complètent les états créés en cours de route. Quand la pile est vide, `stack__pop` déclenche l'exception `stack__Empty`, qui fait sortir de la boucle. L'automate déterministe est alors complet ; il n'y a plus qu'à renvoyer son état initial.

16.6 Réalisation de la commande `grep`

Après cette envolée dans le monde des automates, il est temps de revenir sur Terre et de finir l'implémentation de la commande `grep`. La commande `grep` prend en arguments une expression rationnelle et une liste de noms de fichiers, et affiche toutes les lignes des fichiers qui sont reconnues par l'expression rationnelle. Si aucun nom de fichier n'est fourni, `grep` lit son entrée standard. C'est ce comportement que nous allons maintenant programmer.

Les versions de `grep` qu'on trouve dans les systèmes Unix proposent un certain nombre d'options qui modifient le comportement de la commande : inverser la recherche (afficher uniquement les lignes qui ne sont pas reconnues), identifier majuscules et minuscules, afficher uniquement le nombre de lignes reconnues, etc. Ces options sont faciles mais fastidieuses à implémenter ; nous les laisserons de côté.

Commençons par deux fonctions qui appliquent un automate sur chaque ligne d'un fichier et affichent les lignes reconnues.

Fichier `grep.ml`

```
#open "expr";;
#open "auto";;
#open "determ";;

let ligne_trouvée = ref false;;

let grep_sur_canal auto nom_fich canal =
  try
    while true do
      let ligne = input_line canal in
      if reconnaît auto ligne then begin
        ligne_trouvée := true;
        print_string nom_fich;
        print_string": ";
        print_endline ligne
      end
    done
  with End_of_file -> ();;
```

```
let grep_sur_fichier auto nom_fich =
  try
    let canal = open_in nom_fich in
    try grep_sur_canal auto nom_fich canal; close_in canal
    with exc -> close_in canal; raise exc
  with sys__Sys_error message ->
    prerr_string "Erreur sur le fichier ";
    prerr_string nom_fich;
    prerr_string ": ";
    prerr_endline message;;
```

La dernière phrase du module `grep` vérifie que la ligne de commande a la bonne forme, compile l'expression rationnelle en automate déterministe, et applique l'automate obtenu sur les fichiers spécifiés.

Fichier `grep.ml`

```
if sys__interactive then () else
  if vect_length sys__command_line < 2 then begin
    prerr_endline "Utilisation: grep <motif> <fichiers>";
    exit 2
  end else begin
    let expr =
      try lire (stream_of_string sys__command_line.(1))
      with Parse_error | Parse_failure ->
        prerr_endline "Erreur de syntaxe dans l'expression";
        exit 2 in
    let auto =
      déterminise(expr_vers_automate expr) in
    if vect_length sys__command_line >= 3 then
      for i = 2 to vect_length sys__command_line - 1 do
        grep_sur_fichier auto sys__command_line.(i)
      done
    else
      grep_sur_canal auto "(entrée standard)" std_in;
    exit (if !ligne_trouvée then 0 else 1)
  end;;
```

16.7 Annexe

Pour achever le programme `grep`, il reste à implémenter le module `ensent` qui définit le type abstrait des ensembles d'entiers. En voici une implémentation simple, à base de listes croissantes d'entiers.

Fichier `ensent.ml`

```
type t == int list;;
let vide = [];;
let rec appartient n = function
  | [] -> false
  | m::reste ->
```

```
      if m = n then true else
        if m > n then false else appartient n reste;;

let rec ajoute n = function
  | [] -> [n]
  | m::reste as ens ->
      if m = n then ens else
        if m > n then n :: ens else m :: ajoute n reste;;
```

Comme la liste est triée par ordre croissant, on arrête la recherche ou l'insertion dès qu'on atteint un élément plus grand que l'entier à rechercher ou à insérer. L'insertion et la recherche sont donc en temps moyen $n/2$ et en temps le pire n, si n est le nombre d'éléments de l'ensemble.

16.8 Mise en pratique

Il ne reste plus qu'à compiler tous les modules de la commande `grep` et à les lier entre eux.

```
$ camlc -c expr.mli
$ camlc -c expr.ml
$ camlc -c auto.mli
$ camlc -c auto.ml
$ camlc -c ensent.mli
$ camlc -c ensent.ml
$ camlc -c determ.mli
$ camlc -c determ.ml
$ camlc -c grep.ml
$ camlc -o grep expr.zo auto.zo ensent.zo determ.zo grep.zo
```

En guise d'exemple, voici comment rechercher tous les mots qui contiennent la lettre « p » suivie de la lettre « x » dans un fichier :

```
$ camlrun grep '[pP][a-z]*x' fichier
```

L'exécution de cette commande sur le texte de ce chapitre détecte quatre occurrences du mot « postfixe » et deux de « prix ».

16.9 Pour aller plus loin

La rapidité d'exécution de la commande `grep` implémentée dans ce chapitre pourrait être fortement améliorée. Il y a deux sources importantes d'inefficacité : la déterminisation de l'automate, d'une part, et d'autre part l'exécution de l'automate sur les lignes des fichiers.

La déterminisation est un processus essentiellement coûteux : dans le pire des cas, la taille de l'automate produit est exponentielle en la taille de l'expression rationnelle. Dans les cas courants, on l'accélère considérablement en groupant les transitions sortant d'un état et aboutissant sur le même état. Par exemple, les transitions que nous avons étiquetées « tout sauf `t` », « tout sauf `e` » dans les exemples représentent en fait 255 transitions du même état vers le même état. La fonction de déterminisation présentée

ci-dessus ne tient pas compte de ce genre de partage, et donc a tendance à refaire 255 fois les mêmes calculs dans des situations de ce type. L'introduction, dans les automates non déterministes, de transitions de la forme « tout sauf ... » permet d'éviter cette source d'inefficacité, au prix de nombreuses complications dans le programme.

Pour ce qui est de l'exécution de l'automate déterministe, le problème n'est pas d'ordre algorithmique, mais provient du système Caml Light lui-même : il faut exécuter quelque chose de très simple (une boucle, essentiellement) sur un grand volume de données ; à ce petit jeu, Caml Light se révèle nettement plus lent que des compilateurs traditionnels (Caml Light produit du code pour une machine virtuelle, code qui est ensuite interprété, alors que les compilateurs traditionnels produisent du code directement exécutable par la machine). En ce cas, on gagne beaucoup à utiliser un compilateur Caml produisant du code machine optimisé, tel que le compilateur Objective Caml.

Bibliographie

Pour une bonne présentation des automates vus sous un angle pratique, on se reportera au chapitre 3 de *Compilateurs : principes, techniques et outils*, de Aho, Sethi et Ullman (InterÉditions). Pour les lecteurs que les mathématiques n'effraient pas, signalons que les automates ont beaucoup été étudiés dans le cadre des langages formels, une des branches les plus anciennes de l'informatique théorique. Le livre de Aho et Ullman, *Theory of parsing, translation and compiling : 1 : parsing* (Addison-Wesley), en donne une bonne vue d'ensemble.

III

Introspection

17
Exécution d'un langage fonctionnel

Quand Caml se regarde le nombril, ou commence à tenir ses lacets de chaussures.

Forts de l'expérience acquise avec l'implémentation du mini-Pascal (chapitre 15), nous entamons ici la description et l'implémentation d'un langage fonctionnel simplifié, qui est en réalité un sous-ensemble de Caml et que nous nommerons mini-Caml. Comme d'habitude, nous essaierons de ne pas éluder les difficultés : mini-Caml présente toutes les particularités essentielles de Caml (pleine fonctionnalité, filtrage, polymorphisme). Cette étude nous donnera également l'occasion d'éclaircir un certain nombre de points délicats du langage Caml lui-même, aussi bien dans le domaine de l'exécution des programmes que dans celui de la synthèse des types. Ces points délicats se manifestent rarement lorsqu'on programme en Caml (nous ne les avons pas encore rencontrés dans ce livre), mais apparaissent nettement lorsqu'on implémente Caml.

Dans ce chapitre, nous commençons notre étude par la réalisation d'un interpréteur mini-Caml non typé. Le prochain chapitre aborde la synthèse et la vérification statique des types.

17.1 Le langage mini-Caml

Comme d'habitude, définissons d'abord la syntaxe abstraite du langage mini-Caml que nous étudions. Comme on le voit ci-dessous, nous n'avons conservé que les constructions essentielles de Caml : l'accès à un identificateur, la définition de fonctions par filtrage, l'application de fonctions et la liaison `let`, récursive ou non. S'y ajoutent deux types de base, les entiers et les booléens, et deux structures de données, les paires et les listes.

Fichier `syntaxe.mli`

```
type expression =
   | Variable of string
   | Fonction of (motif * expression) list
```

```
   | Application of expression * expression
   | Let of définition * expression
   | Booléen of bool
   | Nombre of int
   | Paire of expression * expression
   | Nil
   | Cons of expression * expression

and motif =
  | Motif_variable of string
  | Motif_booléen of bool
  | Motif_nombre of int
  | Motif_paire of motif * motif
  | Motif_nil
  | Motif_cons of motif * motif

and définition =
  { récursive: bool;
    nom: string;
    expr: expression };;
```

La pauvreté apparente de ce langage est compensée par le fait que de nombreuses constructions de Caml sont dérivées des constructions de mini-Caml. Par exemple, la construction `match` e `with` $p_1 \rightarrow e_1 \ldots$ n'est autre qu'une application de fonction (`function` $p_1 \rightarrow e_1 \ldots$)($e$). De même, la construction conditionnelle `if` *cond* `then` e_1 `else` e_2 se ramène à `match` *cond* `with true` $\rightarrow e_1$ | `false` $\rightarrow e_2$. D'autre part, toutes les opérations primitives (opérations arithmétiques par exemple) se présentent sous la forme d'identificateurs prédéfinis.

Une phrase mini-Caml est soit une expression, soit une définition. Contrairement à Caml, nous n'avons pas de déclarations de types.

Fichier `syntaxe.mli`

```
type phrase =
   | Expression of expression
   | Définition of définition;;

value lire_phrase: char stream -> phrase;;
```

L'implémentation du module `syntaxe` est tout entière consacrée à la fonction d'analyse syntaxique `lire_phrase`. L'analyseur syntaxique de mini-Caml utilise exactement les mêmes techniques que celui pour mini-Pascal. Nous ne détaillerons donc pas la fonction `lire_phrase` ici, la repoussant sans vergogne à la fin de ce chapitre pour passer plus vite aux choses sérieuses.

17.2 L'évaluateur

Représentation des valeurs

Notre évaluateur manipule des données très simples du type `valeur`. Ce sont les valeurs utilisées par le langage, aussi bien que les résultats des évaluations. Les valeurs

sont de cinq espèces possibles : des nombres entiers, des booléens, des fonctions, des paires ou des cellules de listes. Les fonctions se divisent en deux classes : les opérations primitives, qui opèrent directement sur le type `valeur`, et les fonctions de l'utilisateur. Les fonctions de l'utilisateur sont représentées par des *fermetures*. Une fermeture est une paire dont la première composante est la définition de la fonction (c'est-à-dire le filtrage qui calcule le résultat de la fonction à partir de son argument) et la seconde composante est l'environnement qui prévalait quand on a définit la fonction. Nous allons voir pourquoi ce codage complexe des valeurs fonctionnelles est adéquat à la définition et à l'exécution des fonctions Caml.

Fichier `eval.mli`

```
#open "syntaxe";;
type valeur =
   | Val_nombre of int
   | Val_booléenne of bool
   | Val_paire of valeur * valeur
   | Val_nil
   | Val_cons of valeur * valeur
   | Val_fermeture of fermeture
   | Val_primitive of valeur -> valeur

and fermeture =
  { définition: (motif * expression) list;
    mutable environnement: environnement }

and environnement == (string * valeur) list;;

value évalue: environnement -> expression -> valeur
  and évalue_définition: environnement -> définition -> environnement
  and imprime_valeur: valeur -> unit;;

exception Erreur of string;;
```

La règle de portée statique

Nous devons donc expliquer pourquoi nous codons les fonctions par des fermetures qui comportent l'environnement de définition de ces fonctions. Ceci est rendu nécessaire par la règle de « portée » des identificateurs en Caml. En effet, tout identificateur est lié à la valeur qui l'a défini. Cette liaison ne dépend pas du temps, mais seulement de l'endroit dans le programme où l'identificateur est défini (c'est pourquoi on parle aussi de portée textuelle ou lexicale). Donnons un exemple de ce comportement : nous définissons la constante `taille`, puis la fonction `ajoute_taille` qui fait référence au nom `taille`, puis nous redéfinissons le nom `taille` avec une autre valeur.

```
# let taille = 1;;
taille : int = 1
# let ajoute_taille x = x + taille;;
ajoute_taille : int -> int = <fun>
# let taille = 2;;
taille : int = 2
```

Le problème est de savoir si la redéfinition de `taille` a modifié la fonction `ajoute_taille`, ou bien si cette fonction ajoute toujours 1 à son argument.

```
# ajoute_taille 0;;
- : int = 1
```

Caml suit bien la règle de portée statique : l'identificateur `taille` du corps de la fonction fait référence à celui qui était défini au moment de la création de la fonction, pas à celui qui est défini quand on appelle la fonction. En portée dynamique, la valeur des identificateurs correspond à leur dernière définition au cours des calculs. La valeur de l'identificateur `taille` aurait donc été modifiée même dans le corps de la fonction `ajoute_taille`. Notre évaluateur simule parfaitement ce comportement statique, en attachant au code des fonctions leur environnement de définition.

Les lecteurs attentifs auront sans doute remarqué qu'une certaine forme de portée dynamique peut être simulée en Caml, à l'aide de références.

```
# let taille = ref 1;;
taille : int ref = ref 1
# let ajoute_taille x = x + !taille;;
ajoute_taille : int -> int = <fun>
# taille := 2;;
- : unit = ()
# ajoute_taille 0;;
- : int = 2
```

La liaison de `taille` à la référence est traitée en portée statique, mais le contenu de la référence est modifiable après la liaison. Cette astuce va même jusqu'à la définition de fonctions récursives sans utiliser `let rec`, à la manière des langages avec portée dynamique.

```
# let fact =
      let rien = ref (function x -> x) in
      let f x = if x = 0 then 1 else x * !rien (x - 1) in rien := f;
      f;;
fact : int -> int = <fun>
# fact 10;;
- : int = 3628800
```

Le code de l'évaluateur

La première partie de l'évaluation est consacrée à l'implémentation du filtrage. Cette opération prend une valeur et un motif, et détermine si la valeur est de la forme indiquée par le motif. Si la réponse est non, elle déclenche l'exception `Échec_filtrage`. Si la réponse est oui, elle renvoie un ensemble de liaisons d'identificateurs (les variables du motif) à des valeurs (les morceaux correspondants de la valeur d'entrée), représentées par une liste de paires (identificateur, valeur).

Fichier `eval.ml`

```
#open "syntaxe";;

exception Échec_filtrage;;
```

```
let rec filtrage valeur motif =
  match (valeur, motif) with
  | (val, Motif_variable id) -> [id, val]
  | (Val_booléenne b1, Motif_booléen b2) ->
      if b1 = b2 then [] else raise Échec_filtrage
  | (Val_nombre i1, Motif_nombre i2) ->
      if i1 = i2 then [] else raise Échec_filtrage
  | (Val_paire(v1, v2), Motif_paire(m1, m2)) ->
      filtrage v1 m1 @ filtrage v2 m2
  | (Val_nil, Motif_nil) -> []
  | (Val_cons(v1, v2), Motif_cons(m1, m2)) ->
      filtrage v1 m1 @ filtrage v2 m2
  | (_, _) -> raise Échec_filtrage;;
```

La fonction d'évaluation d'une expression est remarquablement concise. Détaillons-en les principales clauses. L'environnement d'évaluation est représenté par une liste d'association entre les identificateurs et leurs valeurs. Ceci explique la clause des variables. Pour les fonctions on se contente de créer une fermeture qui emmagasine l'environnement courant (`env`) au moment de l'évaluation de la fonction. Pour l'application d'une fonction à son argument, on évalue fonction et argument, et l'on teste si la fonction renvoyée est bien une fonction, c'est-à-dire soit une fermeture, soit une primitive. Dans le cas d'une primitive, on applique directement la valeur fonctionnelle implémentant la primitive. Dans le cas d'une fonction, on essaye de filtrer la valeur de l'argument par les motifs des différents cas de la fonction. Le premier filtrage qui réussit provoque l'évaluation de l'expression associée, dans un environnement qui est l'environnement contenu dans la fermeture, enrichi par les liaisons effectuées lors du filtrage.

Fichier `eval.ml`

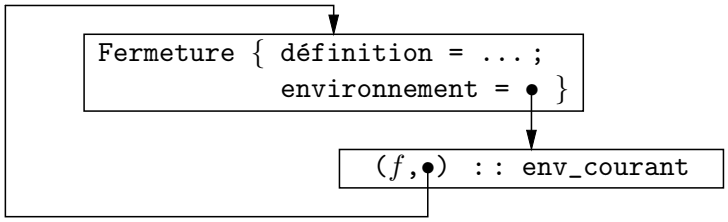
```
let rec évalue env expr =
  match expr with
  | Variable id ->
      begin try
        assoc id env
      with Not_found -> raise(Erreur(id ^ " est inconnu"))
      end
  | Fonction(liste_de_cas) ->
      Val_fermeture {définition = liste_de_cas; environnement = env}
  | Application(fonction, argument) ->
      let val_fonction = évalue env fonction in
      let val_argument = évalue env argument in
      begin match val_fonction with
      | Val_primitive fonction_primitive ->
          fonction_primitive val_argument
      | Val_fermeture fermeture ->
          évalue_application fermeture.environnement
                             fermeture.définition val_argument
      | _ ->
          raise(Erreur "application d'une valeur non fonctionnelle")
      end
```

```
  | Let(définition, corps) ->
      évalue (évalue_définition env définition) corps
  | Booléen b -> Val_booléenne b
  | Nombre n -> Val_nombre n
  | Paire(e1, e2) -> Val_paire(évalue env e1, évalue env e2)
  | Nil -> Val_nil
  | Cons(e1, e2) -> Val_cons(évalue env e1, évalue env e2)

and évalue_application env liste_de_cas argument =
  match liste_de_cas with
  | [] -> raise(Erreur "échec du filtrage")
  | (motif, expr) :: autres_cas ->
      try
        let env_étendu = filtrage argument motif @ env in
        évalue env_étendu expr
      with Échec_filtrage ->
        évalue_application env autres_cas argument

and évalue_définition env_courant déf =
  match déf.récursive with
  | false -> (déf.nom, évalue env_courant déf.expr) :: env_courant
  | true ->
      match déf.expr with
      | Fonction liste_de_cas ->
          let fermeture =
            { définition = liste_de_cas; environnement = [] } in
          let env_étendu =
            (déf.nom, Val_fermeture fermeture) :: env_courant in
          fermeture.environnement <- env_étendu;
          env_étendu
      | _ -> raise(Erreur "let rec non fonctionnel");;
```

Il reste un point délicat à expliquer : la définition récursive. Nous nous sommes limités aux cas où l'expression définissante est une fonction, ce qui garantit que la définition récursive a toujours un sens.

$$\texttt{let rec } f = \texttt{function } p_1 \to e_1 \ldots \texttt{in} \ldots$$

La valeur que nous associons à f est donc une fermeture du corps de la fonction et d'un environnement qui est l'environnement courant d'évaluation augmenté d'une liaison pour f. En effet, le corps de la fonction (l'expression e_1, par exemple) peut faire référence à f, lors d'un appel récursif. L'environnement de la fermeture doit donc contenir une liaison de f à la valeur de f, c'est-à-dire à la fermeture que nous sommes en train de construire. Si nous notons F la fermeture de f, alors l'environnement de cette fermeture doit être `(`f, F`) :: env_courant`. Il est clair que la fermeture F et l'environnement étendu qu'elle comporte sont des objets « circulaires ». Pour résumer graphiquement ces contraintes :



Pour construire la fermeture F, l'astuce est de commencer par construire une fermeture dont le champ `environnement` est initialisé à une valeur quelconque, par exemple l'environnement vide. On construit ensuite l'environnement étendu avec cette valeur provisoire. Il suffit alors de modifier physiquement le champ `environnement` de F pour y stocker l'environnement étendu. La modification physique construit le cycle désiré.

Impression des valeurs

Nous terminons le module `eval` par une fonction d'impression des valeurs, qui ne présente aucune difficulté.

Fichier `eval.ml`

```
let rec imprime_valeur = function
  | Val_nombre n -> print_int n
  | Val_booléenne false -> print_string "false"
  | Val_booléenne true -> print_string "true"
  | Val_paire(v1, v2) ->
      print_string "("; imprime_valeur v1;
      print_string ", "; imprime_valeur v2;
      print_string ")"
  | Val_nil ->
      print_string "[]"
  | Val_cons(v1, v2) ->
      imprime_valeur v1;
      print_string "::"; imprime_valeur v2
  | Val_fermeture _ | Val_primitive _ ->
      print_string "<fun>";;
```

17.3 La boucle d'interaction

Nous allons maintenant mettre autour de l'évaluateur une boucle d'interaction, sur le modèle de la boucle d'interaction de Caml. Nous commençons par construire un environnement initial d'évaluation comprenant un certain nombre de fonctions de base sur les entiers et les booléens (arithmétique, comparaisons, etc.). Pour ce faire, il faut « habiller » les fonctions Caml correspondantes pour qu'elles opèrent non plus sur les types de données Caml, mais sur leurs représentations dans le type `valeur`.

Fichier `interprete.ml`

```
#open "syntaxe";;
#open "eval";;

let code_nombre n = Val_nombre n
```

```
and décode_nombre = function
  | Val_nombre n -> n
  | _ -> raise (Erreur "entier attendu")
and code_booléen b = Val_booléenne b
and décode_booléen = function
  | Val_booléenne b -> b
  | _ -> raise (Erreur "booléen attendu");;

(* Pour transformer une fonction Caml en valeur fonctionnelle *)

let prim1 codeur calcul décodeur =
  Val_primitive(function val -> codeur(calcul(décodeur val)))
and prim2 codeur calcul décodeur1 décodeur2 =
  Val_primitive(function
  | Val_paire(v1, v2) ->
      codeur(calcul (décodeur1 v1) (décodeur2 v2))
  | _ -> raise(Erreur "paire attendue"));;

(* L'environnement initial *)

let env_initial =
  ["+",  prim2 code_nombre  (prefix + ) décode_nombre décode_nombre;
   "-",  prim2 code_nombre  (prefix - ) décode_nombre décode_nombre;
   "*",  prim2 code_nombre  (prefix * ) décode_nombre décode_nombre;
   "/",  prim2 code_nombre  (prefix / ) décode_nombre décode_nombre;
   "=",  prim2 code_booléen (prefix = ) décode_nombre décode_nombre;
   "<>", prim2 code_booléen (prefix <>) décode_nombre décode_nombre;
   "<",  prim2 code_booléen (prefix < ) décode_nombre décode_nombre;
   ">",  prim2 code_booléen (prefix > ) décode_nombre décode_nombre;
   "<=", prim2 code_booléen (prefix <=) décode_nombre décode_nombre;
   ">=", prim2 code_booléen (prefix >=) décode_nombre décode_nombre;
   "not", prim1 code_booléen (prefix not) décode_booléen;
   "read_int", prim1 code_nombre
                     (fun x -> read_int ()) décode_nombre;
   "write_int", prim1 code_nombre
                      (fun x -> print_int x; print_newline (); 0)
                      décode_nombre];;
```

L'évaluation d'une phrase consiste à calculer sa valeur et à l'afficher. Si la phrase est une définition, il faut de plus enrichir l'environnement global par la nouvelle liaison.

Fichier `interprete.ml`

```
let boucle () =
  let env_global = ref env_initial in
  let flux_d'entrée = stream_of_channel std_in in
  while true do
    print_string "# "; flush std_out;
    try
      match lire_phrase flux_d'entrée with
      | Expression expr ->
          let rés = évalue !env_global expr in
          print_string "- = "; imprime_valeur rés;
```

```
          print_newline ()
      | Définition déf ->
          let nouvel_env = évalue_définition !env_global déf in
          begin match nouvel_env with
          | (nom, val) :: _ ->
              print_string nom; print_string " = ";
              imprime_valeur val; print_newline ()
          end;
          env_global := nouvel_env
    with
    | Parse_error | Parse_failure ->
        print_string "Erreur de syntaxe"; print_newline ()
    | Erreur msg ->
        print_string "Erreur à l'évaluation: "; print_string msg;
        print_newline ()
  done;;

if sys__interactive then () else boucle ();;
```

17.4 Mise en œuvre

L'interprète mini-Caml se compile comme suit :

```
$ camlc -c syntaxe.mli
$ camlc -c eval.mli
$ camlc -c eval.ml
$ camlc -c lexuniv.mli
$ camlc -c lexuniv.ml
$ camlc -c syntaxe.ml
$ camlc -c interprete.ml
$ camlc -o interprete lexuniv.zo syntaxe.zo eval.zo interprete.zo
```

Après lancement par `camlrun interprete` ou `interprete__boucle ()`, il ne reste plus qu'à essayer quelques-unes de nos fonctions préférées.

```
# let rec fib = function n ->
    match n < 2 with true -> 1 | false -> fib(n - 1) + fib(n - 2);;
fib = <fun>
# fib 10;;
- = 89
# let map = function f ->
    let rec maprec = function [] -> [] | x :: l -> f x :: maprec l
    in maprec;;
map = <fun>
# map fib (1::2::3::4::5::6::[]);;
- = 1::2::3::5::8::13::[]
```

17.5 Pour aller plus loin

Les modes d'évaluation

Vous aurez sans doute remarqué que notre évaluateur calcule les arguments d'une fonction avant de l'appeler. Cela paraît naturel, mais en fait ce travail s'avère inutile dans le cas où la fonction n'utilise pas son argument. Pire, il peut se produire que le calcul de l'argument ne termine pas, alors même qu'il n'est pas utilisé pour obtenir le résultat final. Il arrive donc que notre évaluateur ne parvienne pas à calculer un résultat pourtant raisonnable. La méthode suivie par notre évaluateur est appelée l'*appel par valeur*, puisqu'on appelle les fonctions après avoir calculé la valeur de tous leurs arguments. La stratégie qui consiste à attendre que le besoin de calculer l'argument se fasse expressément sentir dans le corps de la fonction s'appelle l'*appel par nom*. Son inconvénient majeur est qu'un même argument est calculé plusieurs fois, s'il est utilisé plusieurs fois dans le corps de la fonction. C'est pourquoi il existe un troisième mode d'appel des fonctions, l'*appel par nécessité*, qui consiste, comme dans l'appel par nom, à attendre que la valeur d'argument soit absolument nécessaire avant de le calculer, mais à mettre en mémoire cette valeur calculée, pour la réutiliser telle quelle à chaque fois qu'on en a besoin. Des langages fonctionnels comme Haskell fonctionnent selon ce mode de passage des paramètres ; on les appelle les langages *paresseux*. Leur avantage est évidemment qu'ils peuvent terminer des calculs quand un langage en appel par valeur bouclerait ou échouerait. D'autre part, ce type de langages permet la manipulation aisée de structures de données potentiellement infinies (on ne calcule, de la structure de données, que la partie strictement nécessaire à l'obtention du résultat final). Un écueil majeur de ces langages est que les effets y sont prohibés : il est en effet très difficile de prédire quand le calcul d'un argument va se déclencher, donc impossible de savoir quand vont avoir lieu les effets de bord qu'il comporte éventuellement. Des mécanismes spéciaux comme par exemple les monades sont donc nécessaires pour traiter les entrées-sorties et les exceptions.

L'implémentation d'un évaluateur pour ce type de langage repose sur la création de fermetures (plus techniquement appelées « suspensions » ou « glaçons ») pour les arguments des fonctions : on enferme ainsi le code qui permet de calculer la valeur avec son environnement de définition ; quand on doit évaluer une variable, on lance alors l'exécution du code de sa fermeture dans l'environnement qui l'accompagne (dégel du glaçon). Dans le cas de l'appel par nécessité, il faut en fait créer une référence qui est mise à jour en fin de dégel. Les opérations primitives testent alors si leurs arguments sont déjà calculés ou non. S'ils ne le sont pas, elles les dégèlent.

Tout comme le langage Caml, notre évaluateur fonctionne donc en appel par valeur, mais sa transformation en évaluateur par nom n'est pas très difficile.

Les définitions récursives

Vous aurez remarqué que nous limitons la définition de valeurs récursives aux fonctions immédiates (c'est-à-dire directement introduites par le mot-clé `function`). Ce point est raisonnable mais discutable, car on peut donner un sens à des définitions de valeurs non fonctionnelles. L'écueil est de prétendre donner aussi un sens à des définitions qui n'en ont pas, par exemple `let rec x = x + 1`.

En fait, une définition récursive se ramène toujours à la recherche du point fixe d'une certaine fonction. En effet, toute définition récursive est de la forme `let rec x = phi(x)`, ce qui signifie donc que la valeur de `x` est un point fixe de la fonction `phi`. Par exemple, pour la définition de la fonction factorielle : `let rec fact = function x -> if x = 0 then 1 else x * fact(x - 1)`, la fonction `phi` correspondante est `function f -> function x -> if x = 0 then 1 else x * f(x - 1)`. En effet, `phi(fact)` vaut exactement `fact`.

On montre que la définition de fonctions s'implémente correctement dans un langage en appel par valeur. En revanche, pour les autres valeurs, la classe des définitions acceptables n'est pas très claire. Certains systèmes Caml autorisent la définition récursive de listes bouclées (`let rec x = 1 :: x`).

Dans le cas des langages paresseux, on montre qu'il est toujours possible de traiter une définition récursive par itération d'une fonction à partir de la valeur « indéfini » : voyons l'idée qui sous-tend ce mécanisme avec la définition de la fonction factorielle. On construit d'abord la fonction représentant l'indéfini et la fonction `phi` dont on cherche un point fixe :

```
# let indéfini x = failwith "indéfini";;
indéfini : 'a -> 'b = <fun>
# let phi f = function x -> if x = 0 then 1 else x * f (x - 1);;
phi : (int -> int) -> int -> int = <fun>
```

Puis on définit les itérations successives de `phi` en partant de la valeur indéfinie :

```
# let fact0 = phi indéfini;;
fact0 : int -> int = <fun>
# let fact1 = phi fact0;;
fact1 : int -> int = <fun>
# let fact2 = phi fact1;;
fact2 : int -> int = <fun>
# let fact3 = phi fact2;;
fact3 : int -> int = <fun>
```

Les fonctions `fact0`, `fact1`, `fact2` et `fact3` sont des approximations successives de la fonction factorielle, de plus en plus définies (c'est-à-dire définies sur un nombre croissant d'entiers) :

Argument	`fact0`	`fact1`	`fact2`	`fact3`
0	1	1	1	1
1	indéfini	1	1	1
2	indéfini	indéfini	2	2
3	indéfini	indéfini	indéfini	6

Dans un langage paresseux, les définitions récursives ainsi traitées par itérations successives sont correctes : si le calcul s'arrête, alors la valeur calculée sera un point fixe de l'équation récursive. Au pire, le calcul du point fixe ne termine pas (cas de `let rec x = x + 1`) ou bien échoue, ce qui se traduit par un résultat indéfini (cas de `let rec x = x`).

Les langages paresseux ont donc de belles propriétés sémantiques, mais quelques inconvénients pratiques dans la programmation de tous les jours. Ils ont de plus une regrettable tendance à l'inefficacité : les arguments de fonctions sont la plupart du

temps nécessaires à la fonction, on perd ainsi beaucoup de temps de calcul à créer des suspensions qu'on exécute presque aussitôt. Dans les compilateurs, cela est tempéré par des analyses de « nécessité », c'est-à-dire de détection des endroits où l'on peut compiler du code qui évalue directement un argument qui sera de toutes façons nécessaire dans la suite du code. Ces analyses sont complexes et très difficiles en présence de fonctions d'ordre supérieur.

17.6 Annexe

Comme promis, nous donnons ici le code de l'analyseur syntaxique pour mini-Caml. La plus grande partie du code est une adaptation directe de l'analyseur syntaxique que nous avions écrit pour le langage mini-Pascal (chapitre 15).

Fichier `syntaxe.ml`

```
#open "lexuniv";;

let est_un_opérateur opérateurs = function
  | MC op -> mem op opérateurs
  | _     -> false;;

let lire_opérateur opérateurs = function
  | [< (stream_check (est_un_opérateur opérateurs)) (MC op) >] -> op;;

let lire_opération lire_base opérateurs =
  let rec lire_reste e1 = function
  | [< (lire_opérateur opérateurs) op;
       lire_base e2;
       (lire_reste (Application(Variable op, Paire(e1, e2)))) e >]
          -> e
  | [< >] -> e1 in
  function [< lire_base e1; (lire_reste e1) e >] -> e;;

let lire_infixe lire_base infixe construire_syntaxe flux =
  let rec lire_début = function
  | [< lire_base e1; (lire_reste e1) e2 >] -> e2
  and lire_reste e1 = function
  | [< (stream_check (function MC op -> op = infixe | _ -> false)) _;
       lire_début e2 >] -> construire_syntaxe e1 e2
  | [< >] -> e1 in
  lire_début flux;;

let rec phrase = function
  | [< définition d; (fin_de_définition d) p; 'MC ";;" >] -> p
  | [< expression e; 'MC ";;" >] -> Expression e
and fin_de_définition d = function
  | [< 'MC "in"; expression e >] -> Expression (Let(d, e))
  | [< >] -> Définition d

and expression = function
  | [< définition d; 'MC "in"; expression e >] -> Let(d, e)
```

```
  | [< 'MC "function"; liste_de_cas liste >] ->
      Fonction(liste)
  | [< 'MC "match"; expression e; 'MC "with";
        liste_de_cas liste >] ->
      Application(Fonction(liste), e)
  | [< expr5 e >] -> e
and expr_simple = function
  | [< 'Entier i >] -> Nombre i
  | [< 'MC "true" >] -> Booléen true
  | [< 'MC "false" >] -> Booléen false
  | [< 'Ident id >] -> Variable id
  | [< 'MC "["; 'MC "]" >] -> Nil
  | [< 'MC "("; expression e; 'MC ")" >] -> e
and expr0 = function
  | [< expr_simple es; (suite_d'applications es) e >] -> e
and suite_d'applications f = function
  | [< expr_simple arg;
       (suite_d'applications (Application(f, arg))) e >] -> e
  | [<>] -> f
and expr1 flux =
  lire_opération expr0 ["*"; "/"] flux
and expr2 flux =
  lire_opération expr1 ["+"; "-"] flux
and expr3 flux =
  lire_opération expr2 ["="; "<>"; "<"; ">"; "<="; ">="] flux
and expr4 flux =
  lire_infixe expr3 "::" (fun e1 e2 -> Cons(e1, e2)) flux
and expr5 flux =
  lire_infixe expr4 "," (fun e1 e2 -> Paire(e1, e2)) flux

and définition = function
  | [< 'MC "let"; récursive r; 'Ident nom; 'MC "="; expression e >] ->
      {récursive = r; nom = nom; expr = e}
and récursive = function
  | [< 'MC "rec" >] -> true
  | [< >] -> false

and liste_de_cas = function
  | [< motif m; 'MC "->"; expression e; autres_cas reste >] ->
      (m, e) :: reste
and autres_cas = function
  | [< 'MC "|"; motif m; 'MC "->"; expression e;
       autres_cas reste >] -> (m, e) :: reste
  | [< >] -> []

and motif_simple = function
  | [< 'Ident id >] -> Motif_variable id
  | [< 'Entier n >] -> Motif_nombre n
  | [< 'MC "true" >] -> Motif_booléen true
  | [< 'MC "false" >] -> Motif_booléen false
  | [< 'MC "["; 'MC "]" >] -> Motif_nil
```

```
  | [< 'MC "("; motif e; 'MC ")" >] -> e
and motif1 flux =
  lire_infixe motif_simple "::" (fun m1 m2 -> Motif_cons(m1,m2)) flux
and motif flux =
  lire_infixe motif1 "," (fun m1 m2 -> Motif_paire(m1,m2)) flux;;

let analyseur_lexical = construire_analyseur
   ["function"; "let"; "rec"; "in"; "match"; "with"; "->"; ";;";
    "true"; "false"; "["; "]"; "("; ")"; "::"; "|"; ",";
    "*"; "/"; "-"; "+"; "="; "<>"; "<"; ">"; "<="; ">="; "::"];;

let lire_phrase f = phrase (analyseur_lexical f);;
```

18
Un synthétiseur de types

Thèse : le typage est une aide importante pour le programmeur.
Antithèse : mettre les types à la main dans les programmes, c'est lourd.
Synthèse : utilisons la synthèse automatique de types !

Après l'interprète mini-Caml, nous passons maintenant à la synthèse de types pour ce langage. Vous apprendrez ainsi comment fonctionne le contrôleur de type de Caml. Cette connaissance vous permettra de mieux comprendre les erreurs de typage qu'il vous signale, particulièrement sur les points délicats de polymorphisme et de circularité dans les types. Par-dessus tout, la synthèse de types est un joli problème de manipulations symboliques de données et de résolution de contraintes.

18.1 Principes de la synthèse de types

Les équations entre types

La synthèse de types est analogue à la résolution d'équations mathématiques. On aura donc la notion de variables, de constantes, transposée dans le domaine des types : variables de type, types constants. À la différence des mathématiques où le problème est de résoudre un ensemble d'équations données à l'avance, le synthétiseur de type doit découvrir dans le programme qui lui est soumis l'ensemble des équations à résoudre. Ces équations sont cependant très naturelles. Par exemple, si l'on doit typer l'application `f(x)`, on produira les équations :

$$\begin{array}{lcl} \textit{type de } \texttt{f} & = & t_1 \to t_2 \\ \textit{type de } \texttt{x} & = & t_1 \\ \textit{type de } \texttt{f(x)} & = & t_2 \end{array}$$

Ici les inconnues sont des types t_1 et t_2. Ces nouvelles variables seront automatiquement introduites par le synthétiseur de type. On comprend que si chaque application de fonction introduit deux nouvelles inconnues et trois équations supplémentaires, le problème mène vite à un très grand nombre d'inconnues et d'équations. Cependant, l'algorithme de synthèse de type aura le souci de ne pas introduire de nouvelles inconnues inutilement. Par exemple, si l'on sait déjà que `f` a pour type $ty_1 \to ty_2$, on

se contentera d'utiliser les types ty_1 et ty_2 qui interviennent déjà dans le problème. De plus, l'algorithme n'attend pas d'avoir entièrement construit le système d'équations pour commencer à le résoudre : il effectue simultanément l'introduction des équations et leur résolution.

Polymorphisme et schémas de types

Comme en mathématiques, il arrivera que l'ensemble des équations n'ait pas une solution unique, mais qu'il y ait au contraire une infinité de solutions. C'est généralement le cas en mathématiques lorsqu'il y a plus d'inconnues que d'équations. Considérez par exemple le système d'une équation à deux inconnues x et y

$$x + y = 1.$$

Il possède un degré de liberté : si l'on fixe l'une des inconnues l'autre est calculable instantanément. Ce même phénomène se rencontre dans les systèmes d'équations entre types. Soit par exemple le système :

$$\begin{aligned} t_1 &= \texttt{int} \\ t_2 &= t_3 \to t_1 \end{aligned}$$

qui serait engendré par la phrase `let f x = 1`, où t_2 est le type de `f`, t_1 celui de `1` et t_3 celui de `x`. Par remplacement de t_1, nous en déduisons immédiatement

$$\begin{aligned} t_1 &= \texttt{int} \\ t_2 &= t_3 \to \texttt{int} \end{aligned}$$

Il est impossible d'aller plus loin dans les remplacements et pourtant le système possède toujours un degré de liberté : le type t_3 peut être fixé arbitrairement. Dans le monde des types, on fera alors intervenir le *polymorphisme*, en disant que la solution pour t_2 au système d'équations est un *schéma de type*, avec pour paramètre t_3. Ce schéma de type n'est autre qu'un type de la forme $t_3 \to$ `int` valable pour tout les types t_3. C'est ce que l'imprimeur des types de Caml écrirait `'a -> int` et qui signifie *pour tout type* $'a$, $'a \to$ `int`.

```
# let f x = 1;;
f : 'a -> int = <fun>
```

Les types polymorphes sont donc utilisés pour résumer toutes les solutions possibles à un système d'équations entre types qui n'a pas assez de contraintes pour être résolu complètement par des types de base. Les variables de types qui deviennent ainsi des paramètres du schéma de type sont celles qui ne reçoivent jamais de valeur et qui ne sont donc soumises à aucune contrainte dans le programme.

Méthode de résolution

Pour résoudre les systèmes d'équations entre types, on utilise en première approximation le mécanisme classique de remplacement des inconnues dont on connaît déjà la valeur. C'est ce que nous venons de faire dans l'exemple précédent en remplaçant t_1 par la valeur connue `int`. En fait, le mécanisme de résolution est plus général : c'est une méthode de propagation de contraintes d'égalité connue sous le nom de mécanisme d'*unification*. Nous le verrons en détail par la suite.

Pour modéliser les systèmes d'équations nous aurons donc besoin de variables (de type) pour les inconnues du système, de types constants pour les constantes du système et d'opérations entre types comme la flèche `->` ou le produit `*`. Nous aurons également besoin de modéliser les équations elles-mêmes (le signe =, pourrait-on dire) et donc de rendre compte du remplacement d'une variable par sa valeur. Ce remplacement sera complètement automatique car nous utiliserons le partage : toutes les occurrences d'une inconnue dans les équations du système à résoudre seront représentées physiquement par le même objet en mémoire. Remplacer partout l'inconnue par une valeur reviendra simplement à modifier le contenu de l'objet qui représente l'inconnue en y déposant la valeur. Pratiquement, nos variables auront donc deux statuts possibles : elles seront soit des inconnues (n'ayant donc jamais reçu de valeur), soit des variables connues ayant donc une valeur associée. Par exemple pour le système (résolu)

$$\begin{array}{lcl} t_1 & = & \texttt{int} \\ t_2 & = & t_3 \rightarrow \texttt{int} \end{array}$$

t_1 sera une variable connue valant `int`, t_2 une variable connue valant $t_3 \rightarrow$ `int`, tandis que t_3 sera toujours une variable inconnue.

Utilisation des schémas de types

Le polymorphisme est donc modélisé par des schémas de type. Chaque fois qu'on doit utiliser un schéma de type, on se contente d'utiliser le type qui décrit le schéma, avec de nouvelles inconnues. Par exemple, si l'on veut utiliser le schéma *Pour tout type* $'a$, $'a \rightarrow$ `int`, on utilisera le type $ty \rightarrow$ `int` où ty est une nouvelle inconnue. Le schéma de type correspond à l'ensemble de tous les types obtenus en remplaçant $'a$ par un type quelconque. Une fois $'a$ remplacée par une nouvelle inconnue, cette inconnue pourra recevoir n'importe quelle valeur par la suite. Ainsi, la fonction définie par `let f x = 1` a pour schéma de type *Pour tout type* $'a$, $'a \rightarrow$ `int`. Supposons qu'on écrive `(f 1, f "oui")`. On emploiera deux fois le schéma de type de `f`, une fois avec une nouvelle variable t_1 ($t_1 \rightarrow$ `int`), puis avec une autre variable t_2 ($t_2 \rightarrow$ `int`). Une fois cette substitution de nouvelles inconnues dans le schéma de type effectuée, les deux occurrences de `f` sont munies d'un type « comme les autres » (non pas un schéma) et tout se déroule comme avant. Dans notre exemple, l'application de `f` à `1` engendre l'équation $t_1 =$ `int`, tandis que l'application de `f` à la chaîne `"oui"` engendre l'équation $t_2 =$ `string`. Les inconnues t_1 et t_2 reçoivent ainsi une valeur et l'on en déduit facilement que le type de l'expression `(f 1, f "oui")` est `int * int`.

```
# (f 1, f "oui");;
- : int * int = 1, 1
```

Les schémas de types sont donc des artifices utilisés très ponctuellement par l'algorithme de typage : il ne les manipule pas directement pour résoudre les équations. En effet, quand on emploie un schéma de type, on remplace systématiquement ses paramètres par des inconnues « normales ». L'algorithme de résolution ne manipulera donc que des expressions de types comprenant des inconnues, mais non pas des schémas.

Introduction des schémas de types

Nous avons vu que le synthétiseur de types avait deux activités principales : introduire de nouvelles équations et de nouvelles inconnues et résoudre les systèmes d'équations qu'il a lui-même engendrés. Quand faut-il se contenter d'engranger des équations et à quel moment faut-il décider d'entrer dans la phase de résolution ? Une première réponse simple serait : il faut résoudre en fin de phrase. C'est exact, mais insuffisant. En effet, lorsqu'on écrit en Caml

```
let identité x = x in (identité 1, identité "oui");;
```

il faut décider quel est le type de `identité` avant de typer la partie `in`. En effet, si (et c'est le cas ici) le nom défini par « let » possède un type polymorphe (dont certaines variables restent des inconnues), il faut les détecter tout de suite pour pouvoir employer ce nom avec différents types dans la partie `in`. C'est cette résolution partielle de l'ensemble des équations de typage qui rend la synthèse de type difficile.

Il nous faudra donc résoudre les contraintes avant de typer la partie `in` d'une expression `let`. Plus précisément, il nous faudra seulement découvrir le type de l'identificateur défini, au cas où ce type comporterait des paramètres et serait donc polymorphe. Ici intervient un phénomène que nous admettrons, nous contentant de le justifier intuitivement : seules les inconnues introduites pendant le typage de la définition sont susceptibles de devenir des paramètres du type de l'identificateur défini. Les autres inconnues (celles qui apparaissaient dans le système avant le typage de la définition) ne doivent pas devenir des paramètres, car elles peuvent intervenir plus tard dans des contraintes de types engendrées par le reste du programme. Considérons ce programme :

```
function x -> let y = x in x+y
```

Juste avant de typer la partie `in`, les équations de typage sont

$$\textit{type de } \texttt{x} = t_1 \qquad \textit{type de } \texttt{y} = t_1$$

et il serait incorrect de conclure que `y` a le type *Pour tout* $'a$, $'a$, puisque le reste du programme va révéler que $t_1 =$ `int`. En revanche, on démontre que toutes les inconnues qui ont été introduites pendant le typage de la définition et qui sont toujours inconnues à la fin du typage de la définition, ne seront pas modifiées plus tard par ajout de contraintes supplémentaires ; nous sommes donc fondés à mettre un « pour tout » devant ces inconnues, les transformant en paramètres du schéma de type.

Dans le synthétiseur de types, nous aurons donc un mécanisme pour retrouver facilement toutes les inconnues introduites pendant le typage d'une définition. L'idée est simplement d'associer un « âge » aux inconnues, reflétant la date à laquelle elles ont été introduites.

Remarquons que les définitions d'identificateurs (par `let`) sont les seules constructions qui engendrent des schémas de type : c'est pourquoi on dit souvent qu'en Caml *seul le* `let` *donne du polymorphisme*. En particulier, les arguments de fonctions n'ont jamais de type polymorphe. Ceci vient directement de la structure des types manipulés en Caml : il est impossible d'exprimer avec ces types qu'un argument de fonction doit être polymorphe. En effet les schémas de types de Caml sont de la forme *Pour tous types* $'a$, $'b, \ldots$ *Type*, ce qui signifie que tous les paramètres d'un schéma de

type sont placés en tête de ce schéma (quantification prénexe). On ne peut donc pas exprimer le type d'une fonction dont l'argument serait polymorphe (donc lié à un schéma de type) comme celui d'une fonction qui exigerait que son argument soit au moins aussi polymorphe que l'identité : (*Pour tout type* $'a$, $'a \to' a) \to \ldots$ Cela explique pourquoi la fonction suivante est mal typée :

```
# let phi identité = (identité 1, identité "oui");;
Entrée interactive:
>let phi identité = (identité 1, identité "oui");;
>                                          ^^^^^
Cette expression est de type string,
mais est utilisée avec le type int.
```

On pourrait avoir l'idée d'utiliser une construction `let` pour créer une variable polymorphe égale à l'argument de la fonction `phi` :

```
# let phi identité =
      let id = identité in
      (id 1, id "oui");;
Entrée interactive:
>    (id 1, id "oui");;
>              ^^^^^
Cette expression est de type string,
mais est utilisée avec le type int.
```

Cela ne marche pas car le type de `id` est l'inconnue correspondant au type de l'identificateur `identité` ; or cette inconnue a été créée avant la définition de `id`, lors de l'introduction du paramètre de la fonction `phi`. On pourrait encore définir localement une fonction de même sémantique que l'argument de `phi`, en supposant que son type, n'étant plus directement celui de `identité`, deviendrait polymorphe (il s'agit ici d'une η-expansion).

```
# let phi identité =
      let id x = identité x in
      (id 1, id "oui");;
Entrée interactive:
>    (id 1, id "oui");;
>              ^^^^^
Cette expression est de type string,
mais est utilisée avec le type int.
```

Encore une fois, `id` reste monomorphe, car le synthétiseur de type n'a pas généralisé le type de `id`, qui provenait d'une spécialisation du type de `identité`, l'argument de la fonction `phi` : les inconnues créées pour construire le type de `id` sont aussi « vieilles » que le type dont elles proviennent (dans l'algorithme de typage la vieillesse est héréditaire).

Répétons cette règle fondamentale du typage de Caml : *seul le* `let` *donne du polymorphisme*. Cela a des conséquences étranges pour les fonctions anonymes : elles ne sont jamais polymorphes. C'est pourquoi des phrases sémantiquement équivalentes sont susceptibles d'avoir des types différents. On sait par exemple que `(function x -> e2) e1` produit le même résultat que `let x = e1 in e2`. C'est parfaitement vrai en ce qui concerne l'exécution. C'est faux en ce qui concerne le typage, puisque l'identificateur `x` n'est jamais polymorphe dans la version avec `function`, tandis que la version avec `let` l'y autorise. La construction `match` ... `with` met en évidence le même phénomène

puisqu'elle est équivalente à l'application d'une fonction anonyme. Nous comprenons mieux maintenant le typage des phrases équivalentes suivantes :

```
# (function y -> y y) (function x -> x);;
Entrée interactive:
>(function y -> y y) (function x -> x);;
>                    ^
Cette expression est de type 'a -> 'b,
mais est utilisée avec le type 'a.
# let y = function x -> x in y y;;
- : '_a -> '_a = <fun>
# match (function x -> x) with y -> y y;;
Entrée interactive:
>match (function x -> x) with y -> y y;;
>                                   ^
Cette expression est de type 'a -> 'b,
mais est utilisée avec le type 'a.
```

18.2 L'algorithme de synthèse de types

Nous programmons maintenant l'algorithme de synthèse de types proprement dit, la partie du synthétiseur qui examine le programme et produit les équations entre types qui déterminent les types de tous les objets du programme. Nous verrons plus tard comment ces équations sont résolues et comment les expressions de types sont représentées de manière à manipuler efficacement les équations entre types. Pour l'instant, nous nous contentons de donner l'interface du module `types`, qui définit deux types de données abstraits, `type_simple` et `schéma_de_types`, ainsi que toutes les opérations sur ces types dont nous avons besoin.

Fichier `types.mli`

```
type type_simple and schéma_de_types;;

value type_int: type_simple
  and type_bool: type_simple
  and type_flèche: type_simple -> type_simple -> type_simple
  and type_produit: type_simple -> type_simple -> type_simple
  and type_liste: type_simple -> type_simple;;

value nouvelle_inconnue: unit -> type_simple
  and unifie: type_simple -> type_simple -> unit
  and généralisation: type_simple -> schéma_de_types
  and spécialisation: schéma_de_types -> type_simple
  and schéma_trivial: type_simple -> schéma_de_types
  and début_de_définition: unit -> unit
  and fin_de_définition: unit -> unit;;

exception Conflit of type_simple * type_simple
      and Circularité of type_simple * type_simple;;

value imprime_type: type_simple -> unit
```

```
  and imprime_schéma: schéma_de_types -> unit;;
```

Voici une brève description des opérations du module `types`. Nous donnerons plus de détails par la suite, mais pour l'instant il est inutile d'en savoir plus.

`type_int`, `type_bool`	les constantes de types `int` et `bool`.
`type_flèche` t_1 t_2	renvoie la représentation du type t_1 `->` t_2.
`type_produit` t_1 t_2	renvoie la représentation du type t_1 `*` t_2.
`type_liste` t	renvoie la représentation du type t `list`.
`nouvelle_inconnue ()`	crée une nouvelle inconnue de typage.
`unifie` t_1 t_2	enregistre l'équation $t_1 = t_2$ et la résout compte tenu des équations déjà enregistrées.
`Conflit`, `Circularité`	exceptions déclenchées par `unifie` lorsqu'on lui donne une équation qui n'a pas de solution (comme `int` = `bool`).
`généralisation` t	transforme le type t en un schéma de types, avec pour paramètres toutes les inconnues introduites lors du typage de la précédente définition.
`spécialisation` s	transforme le schéma de types s en un type simple, en remplaçant les paramètres du schéma par de nouvelles inconnues de typage.
`schéma_trivial` t	transforme le type t en un schéma de types « trivial », c'est-à-dire sans aucun paramètre. Sert à mélanger dans la même structure d'environnement de « vrais » schémas de types, tels que ceux obtenus par la construction `let`, et des types simples, tels que les types des arguments des fonctions.
`début_de_définition ()`	signale que l'on commence le typage de la partie définition d'un `let`.
`fin_de_définition ()`	signale que l'on sort du typage de la partie définition d'un `let` et qu'on est sur le point de typer la partie `in`.
`imprime_type` t	affiche le type t à l'écran.
`imprime_schéma` s	même chose pour un schéma.

Le module `synthese` fournit deux fonctions, l'une qui détermine le type d'une expression, l'autre qui détermine le type d'une définition. Les deux fonctions sont paramétrées par un environnement de typage, qui associe des schémas de types aux identificateurs libres de l'expression.

Fichier `synthese.mli`

```
#open "syntaxe";;
#open "types";;

type environnement == (string * schéma_de_types) list;;

value type_exp: environnement -> expression -> type_simple
  and type_déf: environnement -> définition -> environnement;;

exception Erreur of string;;
```

L'implémentation du module `synthese` commence par une fonction auxiliaire de typage des motifs, qui sert à déterminer le type d'une fonction.

Fichier `synthese.ml`

```
#open "syntaxe";;
#open "types";;

let rec type_motif env = function
  | Motif_variable id ->
      let ty = nouvelle_inconnue () in
      (ty, (id, schéma_trivial ty) :: env)
  | Motif_booléen b ->
      (type_bool, env)
  | Motif_nombre n ->
      (type_int, env)
  | Motif_paire(m1, m2) ->
      let (ty1, env1) = type_motif env m1 in
      let (ty2, env2) = type_motif env1 m2 in
      (type_produit ty1 ty2, env2)
  | Motif_nil ->
      (type_liste (nouvelle_inconnue ()), env)
  | Motif_cons(m1, m2) ->
      let (ty1, env1) = type_motif env m1 in
      let (ty2, env2) = type_motif env1 m2 in
      unifie (type_liste ty1) ty2;
      (ty2, env2);;
```

La fonction `type_motif` renvoie deux résultats : d'une part, le type du motif (c'est-à-dire le type des valeurs qu'on a le droit de filtrer par ce motif) ; d'autre part, un environnement de typage étendu, associant des inconnues de typage aux variables du motif. Par exemple, le motif `x::t` a pour type t_1 `list`, où t_1 est une inconnue (puisqu'en l'absence d'information sur les utilisations de `x` et de `t`, on ne sait rien de plus sur le type des listes filtrées par ce motif) et on étend l'environnement avec `x` de type t_1 et `t` de type t_1 `list`. La fonction `type_motif` ne présente pas de difficultés majeures. Le seul cas qui introduit une contrainte de typage est le cas des motifs « cons » $m_1 :: m_2$. Dans ce cas, il faut imposer que le type de m_2 soit un type liste dont les éléments ont pour type celui du motif m_1.

Nous passons maintenant au typage des expressions. Commentons brièvement les cas intéressants. Pour une variable, on va chercher son schéma de types dans l'environnement de typage et on « spécialise » ce schéma en remplaçant ses paramètres par de nouvelles inconnues. Pour une fonction, on type successivement chacun des cas du filtrage qui la définit. Les types des motifs doivent être égaux au type de l'argument de la fonction. Les types des expressions associées doivent être égaux au type du résultat de la fonction. Pour chaque cas, la partie expression est typée dans l'environnement courant étendu par le typage du motif. Pour une application, le type de la partie fonction doit être un type flèche t_1 `->` t_2, avec t_1 égal au type de la partie argument ; t_2 nous donne alors le type du résultat de l'application. Pour un `let`, l'essentiel du travail est fait par la fonction `type_déf` que nous expliquons juste après le code. Les autres constructions se typent de manière évidente.

Fichier `synthese.ml`

```
let rec type_exp env = function
  | Variable id ->
      begin try spécialisation (assoc id env)
      with Not_found -> raise(Erreur(id ^ " est inconnu"))
      end
  | Fonction liste_de_cas ->
      let type_argument = nouvelle_inconnue ()
      and type_résultat = nouvelle_inconnue () in
      let type_cas (motif, expr) =
        let (type_motif, env_étendu) = type_motif env motif in
        unifie type_motif type_argument;
        let type_expr = type_exp env_étendu expr in
        unifie type_expr type_résultat in
      do_list type_cas liste_de_cas;
      type_flèche type_argument type_résultat
  | Application(fonction, argument) ->
      let type_fonction = type_exp env fonction in
      let type_argument = type_exp env argument in
      let type_résultat = nouvelle_inconnue () in
      unifie type_fonction (type_flèche type_argument type_résultat);
      type_résultat
  | Let(déf, corps) -> type_exp (type_déf env déf) corps
  | Booléen b -> type_bool
  | Nombre n -> type_int
  | Paire(e1, e2) -> type_produit (type_exp env e1) (type_exp env e2)
  | Nil -> type_liste (nouvelle_inconnue ())
  | Cons(e1, e2) ->
      let type_e1 = type_exp env e1 in
      let type_e2 = type_exp env e2 in
      unifie (type_liste type_e1) type_e2;
      type_e2

and type_déf env déf =
  début_de_définition ();
  let type_expr =
    match déf.récursive with
    | false -> type_exp env déf.expr
    | true ->
        let type_provisoire = nouvelle_inconnue () in
        let type_expr =
          type_exp ((déf.nom, schéma_trivial type_provisoire) :: env)
                   déf.expr in
        unifie type_expr type_provisoire;
        type_expr in
  fin_de_définition ();
  (déf.nom, généralisation type_expr) :: env;;
```

Le typage des définitions sépare nettement le cas récursif du cas ordinaire. Dans le cas ordinaire, on type simplement l'expression qui définit l'identificateur et l'on ajoute le schéma de types correspondant (obtenu par la fonction `généralisation`) à

l'environnement de typage. Les appels à `début_de_définition` et `fin_de_définition` qui entourent le typage de l'expression définissante permettent à `généralisation` de détecter les inconnues qui doivent être généralisées.

Dans le cas récursif, le mécanisme est analogue, mais on prend la précaution de préenregistrer l'identificateur avec une nouvelle inconnue, avant de typer l'expression. On unifie ensuite le type préenregistré avec le type effectivement trouvé pour l'expression définissante. Comme dans le cas des fonctions, l'identificateur défini récursivement est préenregistré avec un type inconnu ; il n'est donc pas polymorphe dans l'expression définissante. Ceci vous explique pourquoi la fonction `identité` reçoit ici un type monomorphe :

```
# let rec identité x = x
  and message s = print_string (identité s);;
identité : string -> string = <fun>
message : string -> unit = <fun>
```

18.3 Représentation des types

Passons à l'implémentation du module `types`. Les types simples et les schémas de types sont représentés comme suit.

Fichier `types.ml`

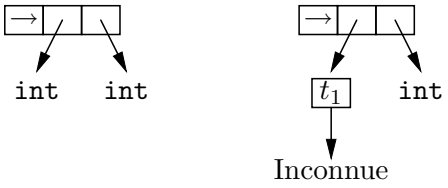
```
type type_simple =
   | Variable of variable_de_type
   | Terme of string * type_simple vect

and variable_de_type =
  { mutable niveau: int;
    mutable valeur: valeur_d'une_variable }

and valeur_d'une_variable =
  | Inconnue
  | Connue of type_simple;;

type schéma_de_types =
  { paramètres: variable_de_type list;
    corps: type_simple };;

let type_int = Terme("int", [||])
and type_bool = Terme("bool", [||])
and type_flèche t1 t2 = Terme("->", [|t1; t2|])
and type_produit t1 t2 = Terme("*", [|t1; t2|])
and type_liste t = Terme("list", [|t|]);;
```

Les types simples sont soit des variables de type, soit des types construits. Une variable de type possède un niveau de liaison, qui identifie la définition où la variable a été créée, et une valeur. Cette valeur est soit inconnue (la variable de type est toujours une inconnue), soit connue ; en ce cas c'est un type simple. Un type construit se compose d'un constructeur de type, comme `int` ou `->`, et le cas échéant des arguments du

constructeur, sous forme d'un tableau de types simples. Les types de base comme `int` et `bool` ont zéro argument, les types listes ont un argument et les types flèches et produits en ont deux.

Pour fournir l'intuition de la méthode de résolution, nous donnons une interprétation graphique de cette méthode. Nous expliquons donc graphiquement comment les équations de typage sont prises en compte grâce à cette représentation des types et des variables de type. Nous représentons les variables de type par des boîtes qui pointent vers une valeur : soit la valeur `Inconnue` si la variable reste une inconnue, soit le type correspondant. Par exemple, dans le schéma suivant, t_1 est une inconnue tandis que t_2 vaut `int`.

Nous représentons les types construits soit simplement par leur nom quand ils ne possèdent pas d'arguments, soit par une boîte comportant des pointeurs vers les arguments du type. Voici les représentations du type `int` $\rightarrow$ `int` et du type $t_1 \rightarrow$ `int` quand t_1 est une inconnue :

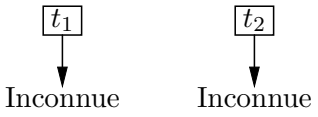


Modification physique directe des variables de type

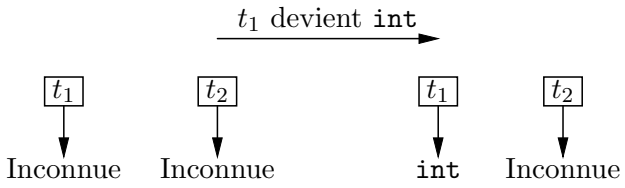
Étudions le déroulement de la résolution du système d'équations :

$$\begin{aligned} t_1 &= \texttt{int} \\ t_2 &= t_1 \rightarrow \texttt{int} \end{aligned}$$

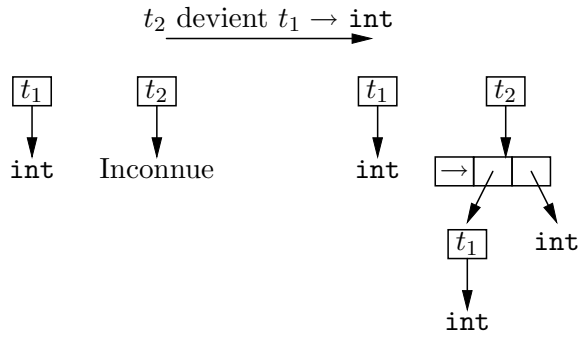
Au départ, nous introduisons les deux inconnues t_1 et t_2.



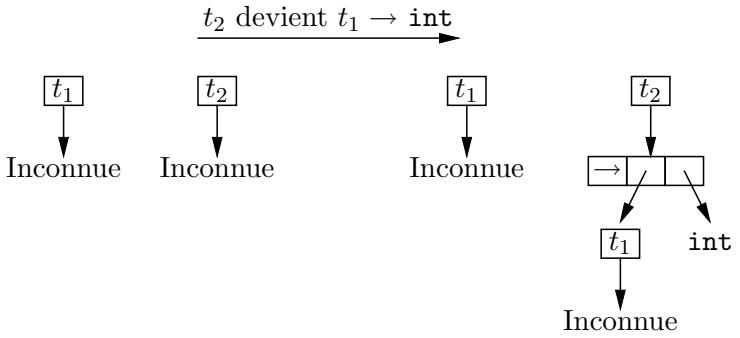
La prise en compte de $t_1 =$ `int` s'effectue par simple modification physique du champ `valeur` de l'inconnue t_1, pour le faire pointer vers le type `int`.



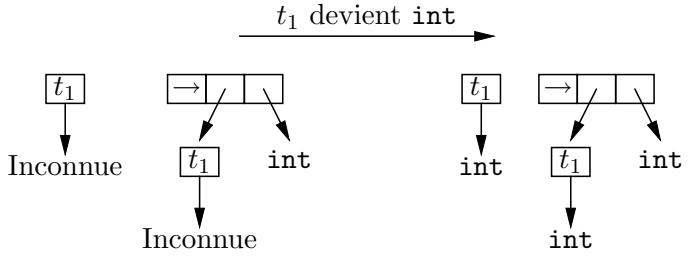
Le traitement de l'équation $t_2 = t_1 \to \texttt{int}$ est similaire.



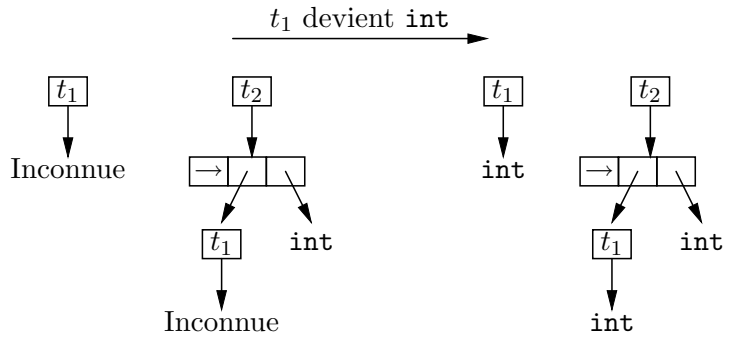
Comme nous l'avons dit, les types sont partagés, ce qui implique que la modification d'un type entraîne automatiquement la modification de tous les types dans lesquels il intervient. Nous allons voir ce mécanisme à l'œuvre dans la résolution du système précédent, en supposant simplement que les deux équations sont présentées dans l'ordre inverse. On commence donc par prendre en compte l'équation $t_2 = t_1 \to \texttt{int}$ et t_1 reste une inconnue.



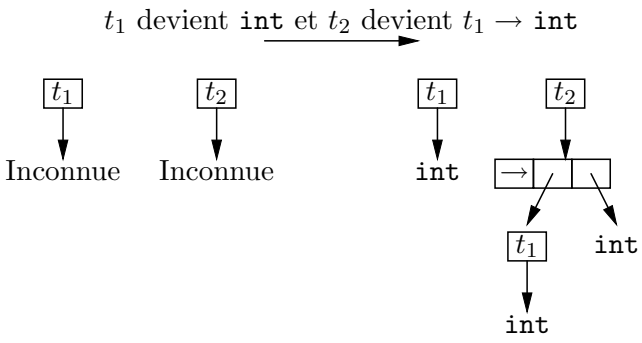
Maintenant, la prise en compte de l'équation $t_1 = \texttt{int}$ modifie automatiquement le type $t_1 \to \texttt{int}$, ce qui a l'effet suivant :



Finalement, grâce au phénomène de partage, la résolution produit exactement le même résultat quel que soit l'ordre dans lequel on résout les équations.



En résumé, la résolution du système produit toujours l'effet suivant :

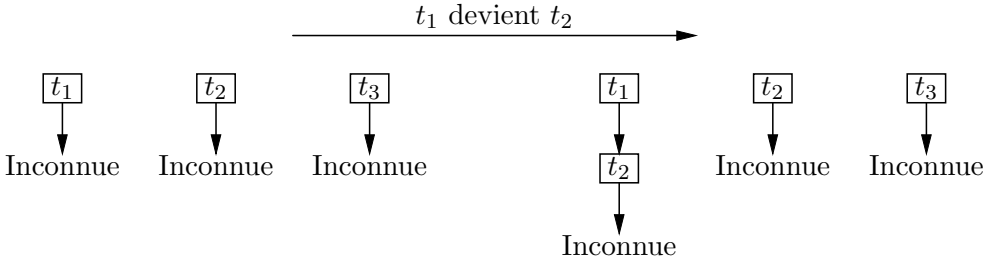


Modification physique des valeurs pointées

Voici un exemple plus difficile, où les modifications physiques doivent s'opérer sur les valeurs pointées par les variables et non sur les variables elles-mêmes.

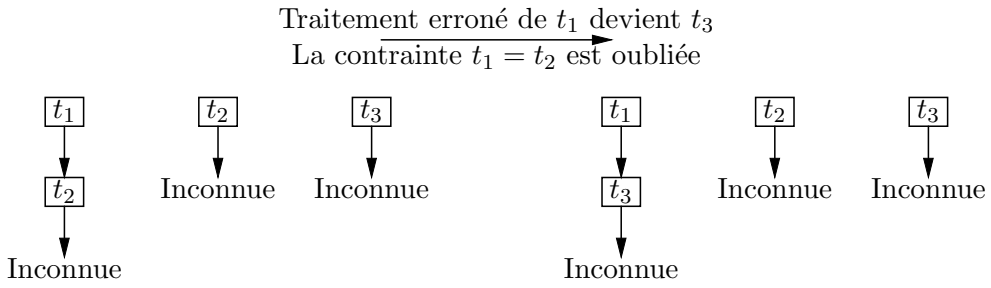
$$\begin{array}{lcl} t_1 & = & t_2 \\ t_1 & = & t_3 \\ t_2 & = & \texttt{int} \end{array}$$

La prise en compte de l'équation $t_1 = t_2$ modifie physiquement la variable t_1, qui pointe maintenant vers t_2. Cette dernière reste une inconnue. t_1 est donc maintenant liée à une autre inconnue.

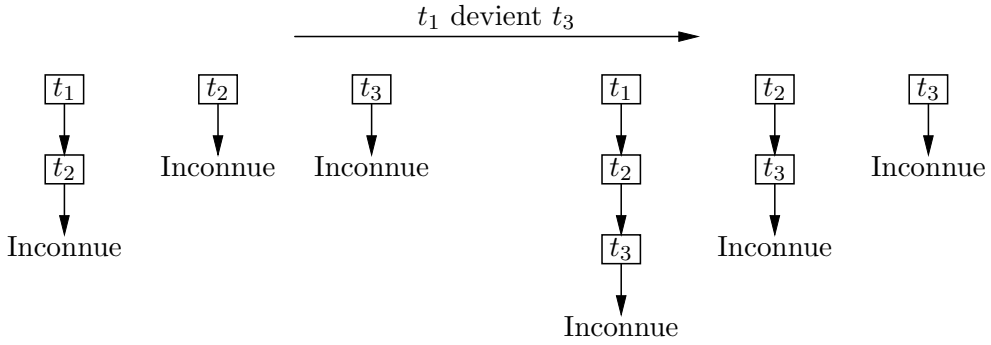


La prise en compte de l'équation $t_1 = t_3$ est plus complexe : puisque t_1 pointe sur t_2, donc que t_1 possède maintenant une valeur, il ne faut surtout pas modifier naïvement

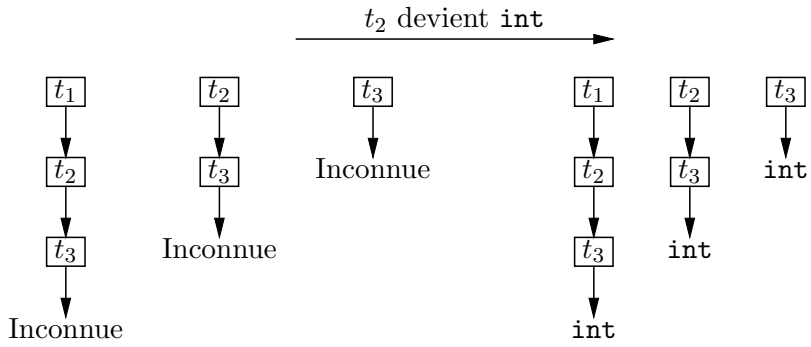
t_1, ce qui aurait pour effet d'oublier la contrainte $t_1 = t_2$ en faisant pointer directement t_1 vers t_3. La modification hâtive de t_1 pour lui donner la valeur de t_3 produirait deux erreurs : premièrement, la contrainte $t_1 = t_2$ serait oubliée au passage, comme nous l'avons dit ; de plus la contrainte induite $t_2 = t_3$ ne serait pas prise en compte, puisque t_2 resterait une inconnue.



Au contraire, il faut suivre le pointeur qui donne la valeur de t_1, ce qui conduit à t_2, et modifier alors t_2. La modification physique correcte est donc la suivante :



La prise en compte de $t_2 =$ `int` est similaire : on modifie simplement la variable vers laquelle pointe t_2, c'est-à-dire t_3, et t_1 est modifiée par effet.

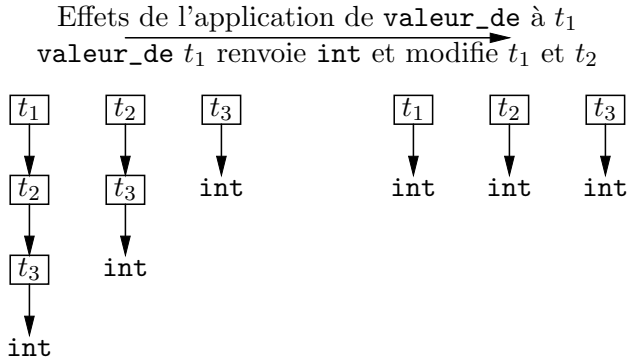


La conclusion de cette discussion est qu'il est généralement incorrect d'opérer sur une variable dont la valeur est connue : il faut directement opérer sur la valeur de cette variable. Bien sûr, si la valeur est elle-même une variable connue, il faut aller chercher la valeur suivante. C'est la tâche de la fonction `valeur_de` : sauter par-dessus les variables connues jusqu'à obtenir soit un terme, soit une variable inconnue.

Fichier `types.ml`

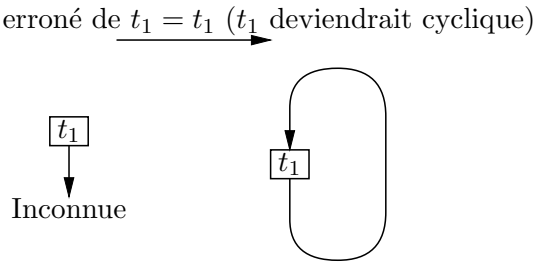
```
let rec valeur_de = function
  | Variable({valeur = Connue ty1} as var) ->
      let valeur_de_ty1 = valeur_de ty1 in
      var.valeur <- Connue valeur_de_ty1;
      valeur_de_ty1
  | ty -> ty;;
```

La fonction `valeur_de` profite de sa recherche de la valeur d'une inconnue pour raccourcir le chemin qui mène à cette valeur. (Le lecteur cultivé aura reconnu la structure de données *union-find* et l'opération de *path compression*.) En reprenant l'exemple des trois variables de types t_1, t_2 et t_3, voici graphiquement l'effet physique d'un appel de `valeur_de` sur la variable t_1 :



Prévention des cycles

Un autre écueil concerne l'équation toute simple $t_1 = t_1$. Pour la prendre en compte, il suffit bien entendu de ne rien faire. Cependant il faut prévoir explicitement ce cas dans les programmes, sous peine de faire pointer la variable de type t_1 vers elle-même et qui plus est de prétendre que t_1 n'est plus une inconnue puisqu'elle a une valeur. Voici ce que donnerait une modification hâtive de t_1 pour la faire pointer vers t_1.



18.4 L'unification

L'unification est le moteur de la résolution des équations de typage. Elle consiste à résoudre un ensemble d'équations, en donnant aux variables de type qui interviennent

dans le problème des valeurs qui rendent toutes les équations vraies. Étant donnée notre représentation des équations, l'unification revient à prendre deux types et à les rendre égaux si nécessaire, en attribuant des valeurs convenables aux inconnues qui apparaissent dans ces types.

Le test d'occurrence

Avant d'aborder l'unification, il nous faut expliquer une dernière subtilité du typage de Caml : le test d'occurrence. Il consiste à vérifier qu'une inconnue n'est pas présente dans un type dont elle doit prendre la valeur. Cela entre dans le cadre de la prévention des cycles, dans un cas de figure plus subtil que l'affectation d'une variable à elle-même. Ce test sert donc à garantir que les types manipulés par l'algorithme de typage sont toujours des arbres ne comportant pas de cycles. En effet, notre algorithme bouclerait si les types qui lui sont soumis devenaient cycliques en cours de typage. Avant de donner à une inconnue la valeur d'un type, on vérifie donc que cette inconnue n'apparaît pas dans le type. Le système Caml fait la même vérification, comme le prouve l'exemple suivant :

```
# let double f = f f;;
Entrée interactive:
>let double f = f f;;
>                  ^
Cette expression est de type 'a -> 'b,
mais est utilisée avec le type 'a.
```

La fonction `test_d'occurrence` prend donc une variable en argument, puis le type qu'on veut lui attribuer et opère une descente récursive dans les arguments de ce type pour vérifier qu'aucun ne contient cette variable.

Fichier `types.ml`

```
let test_d'occurrence var ty =
  let rec test t =
    match valeur_de t with
    | Variable var' ->
        if var == var' then raise(Circularité(Variable var, ty))
    | Terme(constructeur, arguments) ->
        do_vect test arguments
  in test ty;;
```

Pour tester l'égalité entre la variable dont on cherche les occurrences et une autre variable, la fonction `test_d'occurrence` ne doit pas utiliser la fonction d'égalité structurelle de Caml : en effet, toutes les variables qui sont encore inconnues ont la même structure (elles pointent toutes vers le constructeur `Inconnue`) et sont donc structurellement égales. On utilise donc le test d'*égalité physique*, qui indique que ces arguments sont rangés à la même place en mémoire (ce qui assure que ces arguments sont un seul et même objet). L'opérateur de test d'égalité physique de deux valeurs Caml est prédéfini et noté `==` (l'inégalité physique est notée `!=`). Ainsi, quand le test `var == var'` rend vrai, cela indique que `var` et `var'` sont en fait la même variable : dans ce cas on déclenche l'exception `Circularité` avec pour arguments la variable et le type qu'on voulait lui affecter.

Mise à jour des niveaux des variables

Chaque variable porte donc un « niveau », qui indique dans quelle définition elle a été créée. Plus le niveau est élevé, plus la variable a été introduite récemment. Lorsqu'on affecte une variable v par un type t, il faut préserver cette information. En particulier, si le type t contient des variables de niveau plus élevé que v, il faut abaisser le niveau de ces variables au niveau de v. Tout doit se passer comme si, au lieu d'avoir introduit une variable à une certaine date puis déterminé sa valeur par résolution de contraintes, on avait deviné la valeur correcte au moment de l'introduction de la variable. La fonction `rectifie_niveaux` garantit cette propriété.

Fichier `types.ml`

```
let rec rectifie_niveaux niveau_max ty =
  match valeur_de ty with
  | Variable var ->
      if var.niveau > niveau_max then var.niveau <- niveau_max
  | Terme(constructeur, arguments) ->
      do_vect (rectifie_niveaux niveau_max) arguments;;
```

Le moteur de la résolution

La fonction qui unifie deux types procède par filtrage sur ces types et ne rend pas de valeur : elle fait les affectations nécessaires ou bien elle échoue. Elle envisage donc tous les cas possibles de deux valeurs du type `type_simple`. Les deux premiers cas du filtrage concernent le cas d'une variable libre à unifier avec un type, et son cas symétrique. Le dernier cas correspond à deux types construits.

Lorsqu'un des types est une variable, il suffit de modifier physiquement cette variable pour la rendre égale à l'autre type. Comme expliqué ci-dessus, il faut cependant effectuer le test d'occurrence et remettre à jour les niveaux des variables dans le type.

Lorsque les deux types sont des types construits, de deux choses l'une : ou bien leurs constructeurs sont égaux, et alors il suffit d'unifier récursivement leurs arguments pour rendre les deux types égaux ; ou bien leurs constructeurs sont différents, auquel cas l'équation n'a pas de solutions et l'exception `Conflit` est déclenchée.

Il reste un dernier cas, qui est intercepté au tout début de la fonction `unifie` : lorsque les deux types sont déjà égaux, et tout particulièrement lorsqu'ils représentent la même variable, il n'y a rien à faire. Pour détecter cette situation, nous prenons les « valeurs » des deux types, en supprimant les variables connues. Si les valeurs sont physiquement égales, les deux types sont égaux et l'unification s'arrête aussitôt.

Fichier `types.ml`

```
let rec unifie ty1 ty2 =
  let valeur1 = valeur_de ty1
  and valeur2 = valeur_de ty2 in
  if valeur1 == valeur2 then () else
    match (valeur1, valeur2) with
    | Variable var, ty ->
        test_d'occurrence var ty;
        rectifie_niveaux var.niveau ty;
        var.valeur <- Connue ty
```

```
    | ty, Variable var ->
        test_d'occurrence var ty;
        rectifie_niveaux var.niveau ty;
        var.valeur <- Connue ty
    | Terme(constr1, arguments1), Terme(constr2, arguments2) ->
        if constr1 <> constr2 then
          raise (Conflit(valeur1, valeur2))
        else
          for i = 0 to vect_length arguments1 - 1 do
            unifie arguments1.(i) arguments2.(i)
          done;;
```

18.5 Inconnues, généralisation et spécialisation

Le module `type` tient à jour le « niveau de liaison » courant des inconnues. Ce niveau est incrémenté au début du typage d'une définition et décrémenté à la fin. Il mesure la profondeur d'imbrication à gauche des constructions `let`. Les nouvelles inconnues sont créées avec le champ `niveau` égal au niveau courant de liaison.

Fichier `types.ml`

```
let niveau_de_liaison = ref 0;;

let début_de_définition () = incr niveau_de_liaison
and fin_de_définition () = decr niveau_de_liaison;;

let nouvelle_inconnue () =
  Variable {niveau = !niveau_de_liaison; valeur = Inconnue};;
```

L'opération de généralisation consiste à trouver, dans le type à généraliser, toutes les variables dont le niveau est strictement plus grand que le niveau courant de liaison. En supposant qu'on appelle `généralisation` juste après `fin_de_définition`, ces variables sont exactement les inconnues introduites pendant le typage de la dernière définition. La fonction `généralisation` en construit la liste (en faisant bien attention à ne pas mettre plusieurs fois la même variable dans la liste) ; cette liste constitue la liste des paramètres du schéma de type renvoyé par `généralisation`.

Fichier `types.ml`

```
let généralisation ty =
  let params = ref [] in
  let rec trouve_paramètres ty =
    match valeur_de ty with
    | Variable var ->
        if var.niveau > !niveau_de_liaison && not memq var !params
        then params := var :: !params
    | Terme(constr, arguments) ->
        do_vect trouve_paramètres arguments in
  trouve_paramètres ty;
  {paramètres = !params; corps = ty};;
```

```
let schéma_trivial ty = {paramètres = []; corps = ty};;
```

L'opération de spécialisation consiste à associer une nouvelle inconnue à chaque paramètre du schéma, puis à faire une copie du corps du schéma en remplaçant les variables qui sont des paramètres par l'inconnue qui leur est associée. La recherche de l'inconnue associée à un paramètre de type donné a lieu dans la liste d'association `nouvelles_inconnues`, à l'aide de la fonction prédéfinie `assq`, similaire à `assoc` mais utilisant le test d'égalité physique `==` au lieu du test d'égalité structurelle `=`.

Fichier `types.ml`

```
let spécialisation schéma =
  match schéma.paramètres with
  | [] -> schéma.corps
  | params ->
      let nouvelles_inconnues =
        map (fun var -> (var, nouvelle_inconnue ())) params in
      let rec copie ty =
        match valeur_de ty with
        | Variable var as ty ->
            (try assq var nouvelles_inconnues with Not_found -> ty)
        | Terme(constr, arguments) ->
            Terme(constr, map_vect copie arguments) in
      copie schéma.corps;;
```

18.6 Impression des types

Il nous reste à écrire les fonctions d'impression des types. C'est un peu long mais sans réelle difficulté. La seule astuce consiste à produire de jolis noms pour les variables de type (`'a`, `'b`, ...). Nous avons également simplifié le code en supposant que les constructeurs de types ont au plus deux arguments et que ceux qui ont deux arguments se notent de manière infixe (comme `->` et `*`). Ce n'est pas vrai en général, mais c'est le cas en mini-Caml.

Fichier `types.ml`

```
let noms_des_variables = ref ([] : (variable_de_type * string) list)
and compteur_de_variables = ref 0;;

let imprime_var var =
  print_string "'";
  try
    print_string (assq var !noms_des_variables)
  with Not_found ->
    let nom =
      make_string 1
        (char_of_int(int_of_char `a` + !compteur_de_variables)) in
    incr compteur_de_variables;
    noms_des_variables := (var, nom) :: !noms_des_variables;
    print_string nom;;

let rec imprime ty =
```

```
  match valeur_de ty with
  | Variable var ->
      imprime_var var
  | Terme(constructeur, arguments) ->
      match vect_length arguments with
      | 0 -> print_string constructeur
      | 1 -> imprime arguments.(0);
             print_string " "; print_string constructeur
      | 2 -> print_string "("; imprime arguments.(0);
             print_string " "; print_string constructeur;
             print_string " "; imprime arguments.(1);
             print_string ")";;

let imprime_type ty =
  noms_des_variables := [];
  compteur_de_variables := 0;
  imprime ty;;

let imprime_schéma schéma =
  noms_des_variables := [];
  compteur_de_variables := 0;
  if schéma.paramètres <> [] then begin
    print_string "pour tout ";
    do_list (fun var -> imprime_var var; print_string " ")
            schéma.paramètres;
    print_string ", "
  end;
  imprime schéma.corps;;
```

18.7 La boucle d'interaction

Pour finir, il ne nous reste plus qu'à mettre une boucle d'interaction autour du synthétiseur de types. La boucle est un décalque exact de celle utilisée pour l'interprète mini-Caml du chapitre précédent.

Fichier `typeur.ml`

```
#open "syntaxe";;
#open "types";;
#open "synthese";;

let type_arithmétique = schéma_trivial
  (type_flèche (type_produit type_int type_int) type_int)
and type_comparaison =  schéma_trivial
  (type_flèche (type_produit type_int type_int) type_bool);;

let env_initial =
  ["+",  type_arithmétique;     "-",  type_arithmétique;
   "*",  type_arithmétique;     "/",  type_arithmétique;
   "=",  type_comparaison;      "<>", type_comparaison;
   "<",  type_comparaison;      ">",  type_comparaison;
```

```
   "<=", type_comparaison;      ">=", type_comparaison;
   "not", schéma_trivial(type_flèche type_bool type_bool);
   "read_int", schéma_trivial(type_flèche type_int type_int);
   "write_int", schéma_trivial(type_flèche type_int type_int)];;

let boucle () =
  let env_global = ref env_initial in
  let flux_d'entrée = stream_of_channel std_in in
  while true do
    print_string "# "; flush std_out;
    try
      match lire_phrase flux_d'entrée with
      | Expression expr ->
          let ty = type_exp !env_global expr in
          print_string "- : "; imprime_type ty;
          print_newline ()
      | Définition déf ->
          let nouvel_env = type_déf !env_global déf in
          begin match nouvel_env with
          | (nom, schéma) :: _ ->
              print_string nom; print_string " : ";
              imprime_schéma schéma; print_newline ()
          end;
          env_global := nouvel_env
    with
    | Parse_error | Parse_failure ->
        print_string "Erreur de syntaxe"; print_newline ()
    | Conflit(ty1, ty2) ->
        print_string "Incompatibilité de types entre ";
        imprime_type ty1; print_string " et ";
        imprime_type ty2; print_newline ()
    | Circularité(var, ty) ->
        print_string "Impossible d'identifier ";
        imprime_type var; print_string " et ";
        imprime_type ty; print_newline ()
    | Erreur msg ->
        print_string "Erreur de typage: "; print_string msg;
        print_newline ()
  done;;

if sys__interactive then () else boucle ();;
```

18.8 Mise en œuvre

L'ensemble du programme se compile par les commandes suivantes.

```
$ camlc -c syntaxe.mli
$ camlc -c types.mli
$ camlc -c types.ml
$ camlc -c synthese.mli
$ camlc -c synthese.ml
```

```
$ camlc -c lexuniv.mli
$ camlc -c lexuniv.ml
$ camlc -c syntaxe.ml
$ camlc -c typeur.ml
$ camlc -o typeur types.zo synthese.zo lexuniv.zo syntaxe.zo typeur.zo
```

Le programme se lance par `camlrun typeur` ou `typeur__boucle ()` et infère vaillamment les types de nos fonctionnelles préférées.

```
# let rec map =
    function f -> function [] -> [] | a :: l -> f a :: map f l;;
map : pour tout 'a 'b , (('b -> 'a) -> ('b list -> 'a list))
# map (function x -> x + 1) (1 :: 2 :: 3 :: []);;
- : int list
# map (function x -> x) [];;
- : 'a list
# map (function x -> not x) (1 :: 2 :: []);;
Incompatibilité de types entre bool et int
# let id = function x -> x in id (id (id));;
- : ('a -> 'a)
```

18.9 Pour aller plus loin

Complexité de l'algorithme de synthèse de types

Il est très difficile de faire l'analyse de la complexité de l'algorithme de typage que nous avons présenté. C'est si vrai qu'il a fallu attendre plus de dix ans avant que cette étude soit réalisée par P. Kanellakis, H. Mairson et J. Mitchell, dans l'article « Unification and ML type reconstruction », *Computational Logic : Essays in Honor of Alan Robinson*, MIT Press, 1991. Avant cette publication, il était « bien connu » dans la communauté des chercheurs que la synthèse de type d'un langage comme Caml était linéaire. Ce « résultat » purement intuitif venait uniquement de l'expérience, car on « voyait » bien que les compilateurs se comportaient normalement : le temps de compilation doublait approximativement quand la taille du programme doublait.

Le résultat théorique est stupéfiant : le typage des programmes est en fait exponentiel, et même doublement exponentiel dans notre cas. Or nous savons qu'un algorithme exponentiel est catastrophiquement lent quand la taille des données augmente. Le caractère doublement exponentiel de l'algorithme de typage devrait le rendre complètement inutilisable en pratique. Et pourtant, à quelques nuances près, cet algorithme est celui qu'on utilise dans les implémentations de Caml et l'on constate (vous pouvez constater) qu'il est *assez efficace*. La raison en est que le comportement exponentiel, pour indéniable qu'il soit, est extrêmement pathologique. La source de difficulté réside dans le polymorphisme, mais uniquement pour des programmes dont le polymorphisme est extraordinaire, voire déraisonnable.

Pour montrer qu'une phrase peut induire du polymorphisme de façon exponentielle par rapport à sa taille, nous utiliserons la source la plus simple de polymorphisme, la liste vide et la paire. Nous définissons une expression par une cascade de `let ... in`. À chaque étage les paramètres du schéma de type de l'étage précédent sont dupliqués.

Un étage supplémentaire multiplie donc par deux le nombre de paramètres de type du résultat.

```
# let x0 = [] in x0,x0;;
- : 'a list * 'b list = [], []
# let x0 = [] in let x1 = x0,x0 in x1,x1;;
- : ('a list * 'b list) * ('c list * 'd list) = ([], []), ([], [])
```

Plaçons-nous directement au niveau 3 et arrêtons-nous là, parce que les types grossissent trop vite.

```
# let x0 = [] in let x1 = x0,x0 in let x2 = x1,x1 in
    let x3 = x2,x2 in x3,x3;;
- :
 ((('a list * 'b list) * ('c list * 'd list)) *
  (('e list * 'f list) * ('g list * 'h list))) *
 ((('i list * 'j list) * ('k list * 'l list)) *
  (('m list * 'n list) * ('o list * 'p list))) =
 ((([], []), ([], [])), (([], []), ([], []))),
 ((([], []), ([], [])), (([], []), ([], [])))
```

On montre facilement qu'au niveau 10 on aura 2^{10} (soit 1024) paramètres de type et que le type du résultat occupera presque 28000 caractères (15 écrans de 24 lignes sur 80 colonnes). Dans ce cas le typage du programme est bien plus long que son exécution : l'exécution est instantanée, puisqu'il suffit de construire 10 cellules de paires !

La paire (le constructeur infixe « , ») n'est pas responsable de ce phénomène : il est possible de le reproduire en n'utilisant que des fonctions. Il existe en effet un codage fonctionnel de la paire : l'idée consiste à considérer une paire comme un objet qui répond aux messages « première composante » et « seconde composante » ; c'est donc une fonction qui applique une projection à ses deux composantes, à charge pour la projection de sélectionner la composante qui l'intéresse.

```
# let paire x y = function projection -> projection x y;;
paire : 'a -> 'b -> ('a -> 'b -> 'c) -> 'c = <fun>
# let fst paire = paire (function x -> function y -> x)
  and snd paire = paire (function x -> function y -> y);;
fst : (('a -> 'b -> 'a) -> 'c) -> 'c = <fun>
snd : (('a -> 'b -> 'b) -> 'c) -> 'c = <fun>
# let paire_un_true proj = paire 1 true proj;;
paire_un_true : (int -> bool -> 'a) -> 'a = <fun>
# fst paire_un_true;;
- : int = 1
# snd paire_un_true;;
- : bool = true
```

On reproduit alors exactement les mêmes exemples que ci-dessus, sans utiliser de structures de données.

Assez curieusement, la complexité de l'algorithme de typage a deux sources : la recherche du type de la phrase évidemment, mais aussi la simple *impression* du type résultat. On montre en effet que la représentation interne du type des phrases sans `let` est toujours fortement partagée : la taille du type est au plus linéaire par rapport à la taille du programme. Si donc on prend soin de ne pas départager les types pendant

le typage et qu'on écrit les types en exhibant leur partage, l'algorithme de synthèse de type devient linéaire pour les expressions qui ne comportent pas de `let`. Lorsqu'on utilise la construction `let`, il n'existe pas d'algorithme de typage linéaire. La taille du type d'une phrase comportant n constructions `let` est susceptible d'atteindre 2^n. Pire, si l'on ne prend pas soin d'imprimer les types avec le partage, alors la taille du type produit (en nombre de caractères) peut atteindre 2^{2^n} ! Voici un exemple de programme produisant ce comportement pathologique.

```
# let paire x = function proj -> proj x x;;
paire : 'a -> ('a -> 'a -> 'b) -> 'b = <fun>

# let x0 y = paire (paire y);;
x0 : 'a -> (('a -> 'a -> 'b) -> 'b) -> (('a -> 'a -> 'b) -> 'b) -> 'c) ->
'c = <fun>

# let x1 y = x0 (x0 y);;
x1 : 'a -> (((((((('a -> 'a -> 'b) -> 'b) -> (('a -> 'a -> 'b) -> 'b) -> 'c)
-> 'c) -> ((((('a -> 'a -> 'b) -> 'b) -> (('a -> 'a -> 'b) -> 'b) -> 'c) ->
'c) -> 'd) -> 'd) -> ((((((('a -> 'a -> 'b) -> 'b) -> (('a -> 'a -> 'b) -> 'b)
-> 'c) -> 'c) -> ((((('a -> 'a -> 'b) -> 'b) -> (('a -> 'a -> 'b) -> 'b) ->
'c) -> 'c) -> 'd) -> 'd) -> 'e) -> 'e = <fun>
```

Nous n'irons pas plus loin sous peine de remplir ce livre avec les types de cette série. Vous continuerez vous-même avec :

```
let x2 y = x1 (x1 y);; let x3 y = x2 (x2 y);; let x4 y = x3 (x3 y);;
```

À titre indicatif, le type de `x2` dépasse les 72 lignes de 80 caractères, celui de `x3` les 18000 lignes (en fait 1441777 caractères ou environ 300 pages de ce livre !). Pour `x4` nous vous laissons attendre le résultat, s'il vous intéresse ... Retenons qu'il existe des programmes Caml de quelques lignes (mettons trois) qui demandent un temps de typage exorbitant. Nous avons donc la preuve que cet algorithme est au moins exponentiel dans le pire des cas. Sa complexité en moyenne est difficile à estimer (qu'est-ce qu'un programme Caml « moyen » de taille n ?). Nous sommes donc en présence d'un algorithme ayant une complexité extrêmement élevée dans le pire des cas, mais une complexité *linéaire en pratique* (c'est-à-dire pour les données qu'on lui donne effectivement à traiter). Dans le pire des cas, tout se passe comme si on parvenait à soumettre à l'algorithme des données complètement improbables, sur lesquelles il présente une complexité maximale. En pratique, les programmes Caml qu'on écrit vraiment sont peu polymorphes et dans ce cas l'algorithme de typage est effectivement linéaire. Finalement, le pire n'est pas forcément le plus probable, heureusement !

Concept général d'unification

Pour résoudre les équations de typage, nous avons introduit la notion d'unification. Cette notion n'est pas restreinte aux problèmes d'équations entre types : elle se définit dans le cadre plus général des algèbres de termes (c'est-à-dire des structures mathématiques minimales où l'on puisse parler de variables, de constantes et de fonctions d'arité fixée). C'est aussi le mécanisme d'évaluation de base des langages de programmation logique tels que Prolog.

Typage des valeurs mutables

Nous n'avons pas abordé le typage des traits impératifs de Caml. Si la compilation des valeurs mutables et de l'affectation n'est pas un problème difficile, leur typage polymorphe est un problème qui a fait couler beaucoup d'encre. La difficulté vient essentiellement du polymorphisme et de la généralisation des valeurs dont on peut changer dynamiquement le type par affectation, en invalidant ainsi les hypothèses du contrôleur de type. Prenons comme exemple, le cas des références. Le type naturel de `ref` est *Pour tout type* $'a$, $'a \rightarrow$ $'a$ `ref`, celui de l'affectation est *Pour tout type* $'a$, $'a$ `ref` $\rightarrow$ $'a \rightarrow$ `unit` et enfin le déréférencement a pour type *Pour tout type* $'a$, $'a$ `ref` $\rightarrow$ $'a$. Remarquez que ces types sont polymorphes.

Considérez le programme suivant :

```
# let y = ref [] in
      y := true :: !y;
      y := 1 :: !y;
      !y;;
Entrée interactive:
>    y := 1 :: !y;
>               ^^
Cette expression est de type bool list,
mais est utilisée avec le type int list.
```

Le contrôleur de type a très soigneusement évité de généraliser le type de la variable `y`. À défaut, il aurait obtenu le type *Pour tout type* $'a$, $'a$ `list ref`. En ce cas, la première affectation aurait inséré `true` dans la liste pointée par `y` et la seconde aurait été acceptée, insérant un entier dans la même liste. Cela aurait évidemment invalidé l'hypothèse fondamentale que les listes sont homogènes.

Cependant, si le contrôleur de type avait suivi les règles habituelles, cette généralisation aurait dû avoir lieu, comme dans l'exemple similaire sans références.

```
# let y = [] in
  let z = true :: y in
      1 :: y;;
- : int list = [1]
```

De nombreux algorithmes ont été proposés pour typer les valeurs mutables. Tous tentent d'éviter la création de valeurs mutables polymorphes, en restreignant le polymorphisme au niveau de la construction `let`. Nous n'étudierons pas ces algorithmes qui ne sont pas simples et sans doute pas encore définitifs.

Il existe cependant une méthode très simple permettant de régler ce problème : elle consiste à changer l'algorithme de base, bien entendu au niveau du typage du `let`, en décidant que toutes les expressions ne sont pas généralisables : on ne généralise que les constantes, les variables et les fonctions immédiates. La preuve de correction de cet algorithme pour les valeurs mutables est facile à apporter : il n'y a jamais de création de valeurs mutables polymorphes, puisque le polymorphisme est réservé à des expressions qui ne peuvent pas créer de valeurs mutables. C'est en effet clair pour les constantes et les variables. Pour les fonctions immédiates c'est aussi évident : ces fonctions sont celles directement introduites par le mot-clé `function` ; on n'évalue donc rien lors de leur définition.

L'inconvénient de cette méthode est qu'elle modifie l'algorithme de base ; en particulier elle refuse de généraliser les applications, quelles qu'elles soient. Cela interdit de définir une fonction polymorphe par application partielle : `let map_id = map identité;;` est alors typé de façon monomorphe. En pratique, ce n'est pas si grave car il suffit de faire une η-expansion, en ajoutant un paramètre supplémentaire. On écrirait

```
let map_id l = map identité l;;
```

Cette dernière phrase n'est pas vraiment plus complexe que l'application partielle ; on peut même la juger plus claire.

L'avantage fondamental de cette méthode est sa grande simplicité : on conserve les mêmes types qu'avant l'introduction des valeurs mutables et les fonctions manipulant les valeurs mutables sont, sans danger, complètement polymorphes. C'est d'ailleurs la méthode adoptée actuellement dans les compilateurs Caml :

```
# let identité x = x;;
identité : 'a -> 'a = <fun>
# let map_id_poly l = map identité l;;
map_id_poly : 'a list -> 'a list = <fun>
# let map_id = map identité;;
map_id : '_a list -> '_a list = <fun>
```

La variable de type notée `'_a` par le système Caml correspond exactement aux variables de type inconnues de notre contrôleur de type et, comme elles, la variable `'_a` est susceptible de recevoir un type par unification dans la suite du typage du programme :

```
# map_id [1; 2; 3];;
- : int list = [1; 2; 3]
# map_id;;
- : int list -> int list = <fun>
```

La modification de notre contrôleur de type pour qu'il obéisse à cette manière de traiter les valeurs mutables polymorphes est un exercice facile que nous laissons au lecteur.

19
En guise de conclusion

Tout a une fin, mais ce n'est pas triste ...

En conclusion de ce livre, nous aimerions réfléchir sur les idées générales qui se dégagent de l'ensemble des programmes des deuxième et troisième parties de notre ouvrage. Et pour terminer en beauté, nous esquisserons à grands traits ce que pourrait être l'implémentation d'un compilateur Caml, en passant rapidement en revue les principales difficultés spécifiques à la compilation des langages fonctionnels.

19.1 Une méthodologie de programmation

En étudiant les exemples présentés dans ce livre, vous avez pu constater que la démarche était souvent la même : nous définissions d'abord une structure de données, la syntaxe abstraite, puis un moyen commode de faire produire par Caml des valeurs de ce type, la syntaxe concrète avec son analyseur lexico-syntaxique. Après ces deux étapes en guise de préambule, nous passions aux choses sérieuses, à savoir le travail sur la syntaxe abstraite et son interprétation par des programmes d'analyse sémantique. Cette méthodologie a commencé très tôt avec le crayon électronique et le langage mini-Logo et s'est poursuivie ensuite sans discontinuer dans des domaines aussi divers que la démonstration de tautologies avec son langage des formules, la commande `grep` avec son langage d'expressions rationnelles, la pico-machine avec son langage d'assemblage, et bien entendu le mini-Pascal et le mini-Caml dont les langages associés étaient directement des langages de programmation usuels.

Dans tous les cas, nous étions ramenés à définir et implémenter un *langage*, aussi bien en ce qui concerne la syntaxe (abstraite et concrète) que la sémantique. Cette méthodologie est très générale et féconde. Par exemple, un grand nombre de commandes du système d'exploitation Unix se présentent sous la forme de petits langages spécialisés ; c'est également le cas d'éditeurs de textes comme Emacs, de traitements de textes comme TEX, et même des langages HTML et XML de description de pages Web. C'est pourquoi nous avons abondamment illustré cette méthodologie, pour vous permettre de la reconnaître dans les problèmes de programmation qui se présenteront à vous.

Les deux modes d'évaluation

Si le volet syntaxique de cette méthode est relativement invariant d'une application à l'autre, le volet sémantique se subdivise en deux grandes classes : l'interprétation et la compilation. Dans les deux cas, la sémantique consiste en un calcul de valeurs associées aux arbres de syntaxe abstraite. Mais ce calcul s'effectue soit directement, et il s'agit alors d'interprétation (évaluateur des tautologies, du langage graphique, de mini-Caml) ; soit en deux étapes corrélées, en produisant d'abord une nouvelle donnée à partir de l'arbre de syntaxe abstraite, puis en évaluant cette nouvelle donnée, et il s'agit maintenant de compilation (compilation d'un automate à partir d'une expression rationnelle ou production de code pour la pico-machine à partir d'un programme Pascal).

Généralement, les sémantiques à compilateur sont plus efficaces que celles à interpréteur, car la phase de compilation permet d'une part d'anticiper et de préparer la phase d'évaluation et d'autre part de mettre en facteur certaines parties répétitives de l'évaluation. Dans le cas d'un langage de programmation, cet avantage de la compilation est particulièrement clair : lorsque l'interpréteur doit évaluer un programme, il lui faut constamment analyser l'arbre de syntaxe abstraite, alors qu'un code compilé n'a plus de questions à se poser : le compilateur a fait une fois pour toute l'analyse. L'exemple des boucles est frappant à cet égard : l'interprète réanalyse le corps de la boucle à chaque tour de boucle, alors que le code compilé exécute directement ce corps.

Compilation et interactivité

Intéressons-nous plus particulièrement à la dernière partie de ce livre, celle qui concerne l'« introspection » de Caml. Nous avons donné une sémantique à interpréteur pour mini-Caml. Nous l'avons fait à des fins pédagogiques, mais ce n'est pas une méthode réaliste d'implémentation de Caml : tous les systèmes Caml reposent sur des compilateurs. Cela peut surprendre dans la mesure où tous les systèmes Caml proposent aussi une boucle d'interaction à l'utilisateur. Or, il est clair qu'un interprète est plus adapté à l'évaluation interactive, puisqu'il calcule directement la sémantique du programme, alors qu'un compilateur sépare nettement la production du code compilé de son exécution, rendant apparemment impossible l'obtention immédiate du résultat du programme. Cependant, tous les systèmes Caml disposent d'une boucle d'interaction sans interprète, uniquement basée sur un compilateur : chaque phrase entrée par l'utilisateur est aussitôt compilée, puis le code produit est exécuté « à la volée ».

Cette méthode est techniquement plus difficile que l'interprétation, mais elle offre de grands avantages : lorsqu'un langage est évalué par deux méthodes différentes, interprétation et compilation, il se pose immédiatement des problèmes de cohérence entre ces deux méthodes. Il faut en effet prouver que dans tous les cas les résultats produits par l'interprète et par le compilateur sont les mêmes. En ce qui concerne Caml cette propriété est assurée *de facto*, puisqu'il n'y a qu'un seul moyen d'attribuer une sémantique au programme : qu'on soit en programmation séparée ou en interaction directe avec le langage, c'est toujours le même compilateur qui travaille. Pour la même raison, il n'y a pas de différence d'efficacité entre programmes indépendants et pro-

grammes du système interactif : compilés comme les autres, les programmes développés interactivement s'exécutent forcément à la même vitesse.

19.2 La compilation de Caml

En point d'orgue à ce livre, nous aurions naturellement aimé vous présenter un compilateur pour le langage mini-Caml, produisant du code pour la pico-machine. Nous y avons renoncé pour des questions de volume : un tel compilateur est un assez gros programme ; même si la plupart des techniques introduites dans le compilateur mini-Pascal s'appliquent sans changements à mini-Caml, il reste à résoudre un certain nombre de difficultés propres à Caml. La quarantaine de pages de code et d'explications nécessaires auraient rendu ce livre trop épais. Dans cette conclusion, nous nous contenterons de donner un aperçu des problèmes nouveaux que pose la compilation de Caml et des techniques mises en œuvre dans les systèmes Caml.

La gestion automatique de la mémoire

Allocation de mémoire Le langage Caml nécessite des méthodes complexes de gestion de la mémoire, c'est-à-dire d'allocation et de libération des adresses mémoires. L'allocation consiste à réserver un certain espace dans la mémoire (un *bloc* de mémoire) pour y ranger des données. On l'utilise par exemple pour fabriquer des tableaux, des paires, des cellules de listes ou des chaînes de caractères. Un programme Caml ordinaire alloue un grand nombre de blocs, mais ces blocs ont généralement une durée de vie assez brève. Un exemple simple : lorsqu'on utilise des listes, il est courant de créer une liste pour y appliquer aussitôt `map` ; en ce cas, la liste de départ est devenue inutile, car seule la liste résultat est utilisée dans le reste du programme. Cela signifie qu'on remplit facilement toute la mémoire avec des blocs dont beaucoup ne servent plus. Pour éviter cela, il faut savoir libérer les blocs mémoire devenus inutiles, pour pouvoir réutiliser la place mémoire qu'ils occupaient.

Récupération de mémoire Il y a principalement deux moyens de libérer des blocs : le premier consiste simplement à laisser ce soin au programmeur, qui devra explicitement signaler quand un bloc alloué est libre. Le second est la libération implicite, gérée automatiquement par un programme spécialisé, le récupérateur de mémoire. La libération explicite n'existe pas en Caml, car c'est une source d'erreurs subtiles et fréquentes. Fréquentes, parce qu'il est facile d'oublier qu'une partie d'une structure de données est utilisée par la suite et donc de libérer trop tôt cette structure. Subtiles, car lorsqu'un bloc mémoire est libéré, les données qu'il contenait ne sont pas immédiatement détruites : elles resteront valides tant qu'on n'écrira pas d'autres données au même endroit. Cela signifie que le programme continuera à marcher un certain temps après la libération qui crée l'erreur. Au gré du chemin pris dans le programme, ces valeurs seront détruites plus ou moins tard, donnant à l'utilisateur l'impression que son programme se comporte de manière erratique.

La récupération automatique de mémoire La manipulation sûre et facile des structures de données suppose donc l'allocation et la libération automatique des blocs de mémoire. Les programmes Caml s'exécutent donc en collaboration avec un programme spécialisé pour gérer la mémoire de la machine : le gestionnaire mémoire. Ce programme se compose de deux parties, l'allocateur de mémoire et le récupérateur de la mémoire inutilisée. Le récupérateur est communément appelé « GC », pour *garbage collector*, littéralement « éboueur ». On traduit généralement GC par « glaneur de cellules » ou encore « ramasse-miettes ». Le mécanisme général du GC est le suivant : lorsque l'allocateur de mémoire ne peut satisfaire une requête par manque de mémoire libre, le GC se déclenche et parcourt récursivement toutes les données utilisées par le programme en cours. Il commence par le contenu des registres, de la pile et de toutes les variables globales, puis « descend » récursivement dans les structures de données. De ce parcours, le GC déduit l'ensemble des adresses mémoire accessibles, donc potentiellement utilisées par le programme. Toutes les autres adresses sont forcément inutilisées et donc récupérables.

Ce mécanisme de parcours des données actives impose des contraintes sur la représentation des structures de données en mémoire. Essentiellement, le GC doit savoir distinguer, parmi les champs d'une structure, les pointeurs vers des sous-structures (qu'il faut parcourir récursivement) des données qui ne sont pas des pointeurs, comme par exemple les nombres entiers (sur lesquels le GC doit arrêter son parcours). L'approche suivie par les systèmes Caml est de coder pointeurs et entiers de manière à les distinguer par examen de leur code. Par exemple, on code les adresses mémoires par des mots pairs et les entiers par des mots impairs. Le GC « sait » alors qu'il doit parcourir récursivement les mots pairs et ignorer les mots impairs. Toutes les données non entières sont représentées par l'adresse mémoire d'un bloc de mémoire alloué, bloc qui est muni d'un en-tête indiquant au GC la taille du bloc et le type de données contenues dans le bloc. Les adresses sont naturellement des mots pairs sur la plupart des machines. Quant aux entiers, pour garantir qu'ils sont toujours représentés par des mots impairs, on représente l'entier Caml n par le mot $2n+1$ dans la machine. On compile alors sans difficultés les primitives arithmétiques en tenant compte de ce codage (l'addition de deux entiers consiste à additionner leurs codes et à soustraire 1 au résultat, par exemple). Le prix de ce codage est un léger ralentissement des opérations arithmétiques et l'impossibilité d'obtenir tout l'intervalle des entiers représentables par un mot machine (on perd un bit).

La compilation du polymorphisme

Tout comme la récupération automatique de mémoire, le polymorphisme impose également des contraintes sur la représentation des données Caml dans la machine. Les fonctions polymorphes sont appliquées à des données de types différents sur lesquelles ces fonctions opèrent de façon uniforme. Le même code machine doit donc opérer uniformément sur des données de tous les types, des entiers aux structures de données les plus complexes. Ceci n'est possible que si tous les types de données partagent un format commun de représentation ; en particulier, si toutes les représentations ont la même taille. En général on choisit une taille d'un mot mémoire. Les objets qui occupent naturellement plus d'un mot (nombres flottants, n-uplets, etc.) sont alors alloués

en mémoire et manipulés par l'intermédiaire d'un pointeur. Cette approche simple se prête parfaitement au polymorphisme, au prix d'une certaine inefficacité dans les manipulations d'objets alloués.

Pour essayer de réduire cette inefficacité, une autre voie prometteuse a été proposée récemment pour admettre des données de taille hétérogène (comme les tableaux alloués à plat de mini-Pascal), tout en conservant le polymorphisme paramétrique de Caml : il s'agit de modifier la représentation des données au cours de l'exécution des programmes. Ainsi, certaines fonctions non polymorphes travaillent avec des données occupant plus d'un mot (par exemple des flottants sur deux mots), tandis que les fonctions polymorphes travaillent systématiquement avec des données d'un mot. Lorsque des fonctions polymorphes et des fonctions normales échangent des données, celles-ci sont allouées dans un objet structuré à l'entrée des fonctions polymorphes qui ne connaissent pas la représentation spécifique des données ; ainsi les fonctions polymorphes reçoivent toujours un objet de taille fixe, entier ou pointeur sur leur argument. Symétriquement, à la sortie des fonctions polymorphes, les données sont extraites de l'objet structuré qui les contient et remises à plat pour être passées aux fonctions monomorphes qui connaissent leur représentation. Il va sans dire que cette méthode est bien plus complexe que la précédente, mais donne généralement de meilleurs résultats. Actuellement, tous les systèmes Caml fonctionnent avec des données de taille uniforme.

La compilation de la fonctionnalité

Les fermetures Au niveau du compilateur proprement dit, la principale nouveauté de Caml par rapport à Pascal est la pleine fonctionnalité. Lorsque les fonctions peuvent être calculées et renvoyées en résultat, il n'est plus possible de les représenter uniquement par des adresses de morceaux de code machine. Comme nous l'avons vu dans l'interpréteur mini-Caml, il faut introduire la notion de fermeture, c'est-à-dire transformer les fonctions en structures de données allouées contenant, en plus de l'adresse du code de la fonction, l'environnement au moment de la définition de la fonction. Le code produit pour une définition de fonction alloue cette fermeture et y stocke l'environnement courant. Le code produit pour appliquer une fonction va chercher dans la fermeture l'adresse du code de la fonction et se branche à cette adresse, non sans avoir passé la partie environnement de la fermeture en argument supplémentaire à la fonction. Le corps de la fonction est compilé de manière à aller chercher dans cet argument supplémentaire la valeur des identificateurs libres.

Appel de fonctions inconnues En plus du recours aux fermetures, la pleine fonctionnalité impose un mécanisme uniforme d'appel de fonctions. Toute fonction Caml est susceptible d'être appelée par une fonction qui ne connaît rien de la fonction qu'elle appelle : par exemple, `map` reçoit n'importe quelle fonction en argument et l'appelle sans rien savoir à son propos. En général, lorsque le compilateur produit du code pour une application de fonction, il ne connaît pas la définition de cette fonction, ce qui l'empêche de faire certaines optimisations simples. En particulier, une application de fonction à plusieurs arguments $f\ e_1\ \ldots\ e_n$ n'est généralement pas compilable « à la Pascal », en passant les n arguments d'un seul coup, car rien ne garantit que f soit une fonction à n arguments de la forme `f` $x_1\ \ldots\ x_n$ `=` $\ldots$: elle pourrait aussi bien être le résultat

d'un calcul beaucoup plus compliqué, rendant nécessaire de passer les n arguments un par un, avec des constructions de fermetures intermédiaires (pensez à `map successeur`, pour une fonction à un argument, et à `let f x = let ... in (function y -> ...)`, pour une fonction à deux arguments). En bref, la notion d'arité d'une fonction est difficile à définir en Caml : elle ne se détecte pas directement par le type des fonctions et impose donc une analyse parallèle au typage. La difficulté est encore plus grande si l'on souhaite traiter de la même manière les fonctions n-aires curryfiées et les fonctions n-aires non curryfiées (celles dont les arguments sont syntaxiquement sous la forme d'un n-uplet). Les meilleurs compilateurs Caml savent optimiser les appels directs aux deux types de fonctions.

Création des fermetures En pratique, beaucoup de programmes Caml n'utilisent pas la pleine fonctionnalité et sont en fait très proches de programmes Pascal (tout au moins du point de vue de la compilation). Si le compilateur s'efforce d'adopter pour ces programmes la même stratégie qu'un compilateur Pascal, il fera du bon travail. Par exemple, dans les programmes courants, la plupart des appels de fonctions concernent des fonctions connues du compilateur et dans ce cas le compilateur produit du code plus efficace, en passant tous les arguments d'un seul coup et en engendrant un appel direct au code de la fonction. De la même façon, on n'est pas obligé de fabriquer systématiquement une fermeture pour toutes les fonctions d'un programme car beaucoup de fonctions restent locales au module (ou à la phrase) qui les définit. Le compilateur doit s'efforcer de détecter ces cas. Remarquez cependant qu'on ne peut pas éliminer la fabrication *dynamique* de fermetures, au cours de l'exécution (pensez à `let g = function x -> function y -> x + y`, puis à `let h = g 3`, ou encore à des fonctions dont la valeur est écrite dans une structure de données).

Lorsqu'il est contraint et forcé d'allouer une fermeture, le compilateur a le choix entre plusieurs stratégies d'allocation des environnements de fermetures. La première est le partage maximal des environnements : l'environnement d'exécution du programme est constamment maintenu à l'exécution, comme pour notre interpréteur mini-Caml, et l'allocation d'une fermeture se réduit à créer une paire entre l'environnement courant et l'adresse de code de la fonction. L'autre stratégie est l'allocation « à plat » des fermetures. Le compilateur crée un tableau contenant les valeurs des variables libres du corps de la fonction. Cette méthode assure que l'environnement de la fermeture contient seulement les valeurs nécessaires à l'exécution de la fonction, mais elle oblige à recopier ces valeurs à chaque création de fermeture. La première méthode partage au maximum les environnements, mais tout l'environnement d'exécution est mis dans la fermeture créée. Cette méthode met donc aussi le maximum de valeurs inutiles dans les fermetures, occasionnant ainsi des *fuites de mémoires*, ce qui correspond à la rétention de cases mémoires non utilisées mais irrécupérables par le GC, car toujours accessibles à partir des données du programme (dans notre cas, une fermeture qui contient une donnée à laquelle elle n'accédera jamais). Ces fuites de mémoires s'avèrent rédhibitoires pour certains programmes, dans la mesure où elles sont imparables : le programmeur n'a pas les moyens de les éviter, puisque c'est la méthode de compilation des programmes qui les engendre. C'est pourquoi nous préférons l'allocation à plat, qui tient un plus juste compte des objets réellement indispensables à l'exécution.

Le socle du compilateur En généralisant la discussion ci-dessus, il apparaît deux approches radicalement différentes de la compilation de Caml. L'une consiste à s'appuyer sur un modèle d'exécution intégrant la pleine fonctionnalité (dans lequel les fonctions peuvent avoir des variables libres). Ce modèle d'exécution est souvent basé sur une machine virtuelle pour l'exécution du λ-calcul. Les seules optimisations que le compilateur peut alors effectuer sont des transformations de programmes de haut niveau ; l'optimisation des fonctions elles-mêmes et de leur représentation sous forme de fermetures est difficilement exprimable. L'autre approche consiste à exposer beaucoup plus tôt la représentation des fonctions par des fermetures. On se ramène ainsi à un langage intermédiaire de type langage algorithmique classique, souvent proche du langage C, sur lequel le compilateur peut appliquer de nombreuses optimisations de bas niveau. Cette dernière approche, quoique plus complexe, donne généralement de meilleurs résultats : les compilateurs obtenus par cette approche compilent bien ce qu'il est facile de bien compiler (appels à des fonctions connues, fonctions sans variables libres), ce qui représente une large part des programmes qu'on écrit en Caml. Le principal écueil qui menace cette approche est, à force d'améliorer les cas simples, de trop négliger les cas compliqués et d'aboutir à une compilation incorrecte de ces cas difficiles.

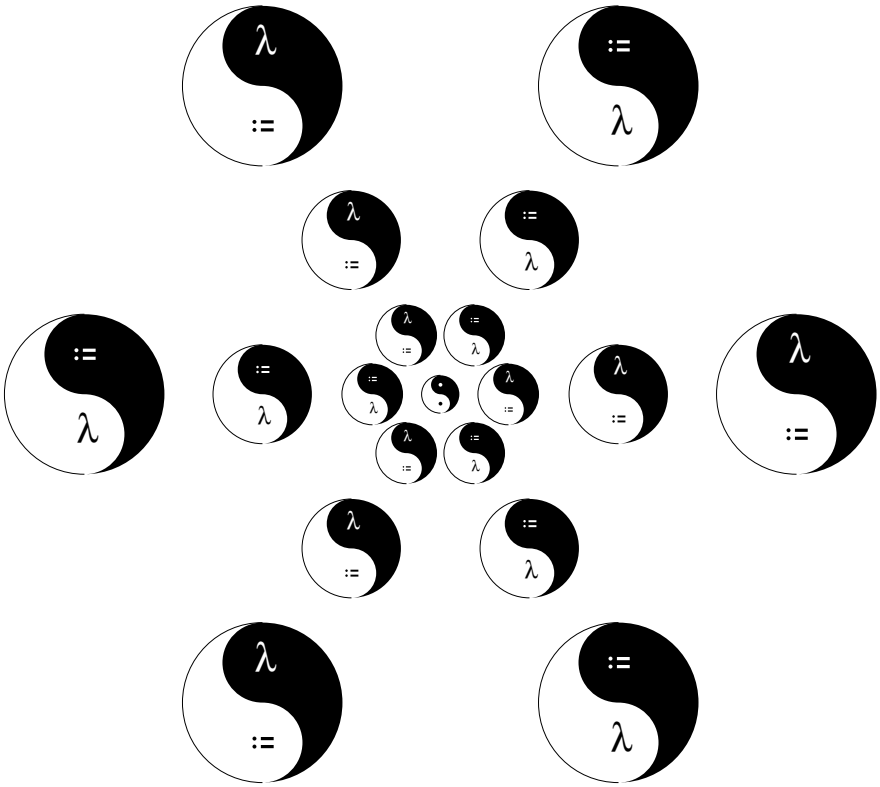
L'auto-génération

Quoi qu'il en soit, vous devez entrevoir maintenant les étapes principales d'une implémentation complète de Caml en Caml : description de la syntaxe abstraite, de la syntaxe concrète, du typage, et enfin de la compilation. Ce mécanisme d'auto-description est général dans les systèmes Caml : il sont tous *autogènes*, c'est-à-dire produits par eux-mêmes. En effet, le compilateur est entièrement écrit en Caml, c'est donc un programme Caml comme tous les autres, compilable par le compilateur Caml, c'est-à-dire par lui-même. Ce mécanisme d'auto-amorçage ou d'auto-génération s'appelle *bootstrap* en anglais.

L'auto-génération est un mécanisme étrange, puisqu'il s'agit d'une sorte de définition récursive au niveau de la spécification exécutable du compilateur du langage. On se demande bien par quel miracle il en sort un système Caml qui tourne. C'est simplement que la récursion s'arrête sur le compilateur « de niveau zéro », le compilateur initial, qui est toujours écrit dans un autre langage. Progressivement, certaines parties du compilateur initial sont réécrites dans le langage compilable par le compilateur, jusqu'à obtenir finalement un compilateur entièrement écrit dans le langage du compilateur : le langage est devenu autogène.

Au-delà du tour de force qu'elle constitue, l'auto-génération est un gage de qualité pour un compilateur : le programme qui réalise la compilation est forcément complexe et long, c'est un bon critère de savoir que le compilateur est capable de le compiler sans erreurs, en produisant un code acceptable en un temps raisonnable. C'est une étape majeure dans la validation d'un langage de programmation et de son implémentation. Rappelons encore une fois que tous les systèmes Caml sont autogènes.

L'auto-génération d'un compilateur Caml serait donc une belle aventure à raconter, « mais ceci est une autre histoire ... »


λ
:=
:=
λ
λ
:=
:=
λ
:=
λ
:=
λ
λ
:=
:=
λ
λ
:=
:=
λ
λ
:=
:=
λ
λ
:=
:=
λ
λ
:=
:=
λ
λ
:=
:=
λ
λ
:=
:=
λ
λ
:=
:=
λ

Index

échec, 77, 126
écriture d'une référence, 48
écriture dans un enregistrement, 119, 150
écriture dans un tableau, 41
écriture dans une chaîne, 46
écriture sur fichier, 182
édition de liens, 187
égalité, 43, 121, 316
égalité physique, 354
énumérés (types), 113
équivalence, 211
étiquette, 117
étiquettes (d'assemblage), 267
étoile, 42
évaluation paresseuse, 162, 334
`!`, 47
`!=`, 354
`"..."`, 5
`#`, 6, 148
`#open`, 184
`&&`, 26, 214
`()`, 14
`(*`, 10
`*`, 63
`*)`, 10
`*.`, 149
`+.`, 149
`->`, 11, 28, 63–65
`.`, 117, 149
`.(`, 41
`.[`, 24, 46, 133, 164
`:`, 11
`::`, 75
`:=`, 48
`;`, 14
`;;`, 6
`<-`, 41, 46, 119, 150, 164
`<>`, 43
`=`, 214, 316
`==`, 354
`@`, 92, 102
`[...]`, 75
`[<...>]`, 162, 163, 165
`[]`, 75
`[|...|]`, 40
`^`, 8, 68, 88, 100
`_`, 28, 85
`` `...` ``, 24
`{...}`, 116
`|`, 28, 110, 135, 163
`||`, 26, 214
`'_a`, 87, 364
`'`, 162
`'a`, 58

abréviation de type, 261
abstraction, 66, 81, 91
accès dans un tableau, 41
accès direct dans les fichiers, 246
accumulateur, 47, 49, 61, 102, 216
affectation, 37, 41, 48, 54
aléatoires (nombres), 140, 201
allocation mémoire, 290, 367
alternative, 12, 26, 43
analyse de cas, 28
analyse lexicale, 160, 162, 217, 228
analyse syntaxique, 159, 160, 166, 173, 218, 271, 279, 308, 336, 365
`and`, 8
appel par nécessité, 334

appel par nom, 334
appel par valeur, 334
appels système (du pico-processeur), 265
application, 9, 16, 73
application partielle, 64, 67, 221, 231
arbres de Huffman, 242
arbres tournoi, 248
argument négatif, 15
arguments d'une commande, 240
`as`, 80, 117
ASCII (code), 133, 163
assembleur, 267
associativité, 16
auto-génération, 371
automates, 309
automates déterministes, 313

`begin`, 14
belote, 115
bien fondée (récursion), 19, 22
`bind`, 197
bit à bit (opérations), 250
bloc d'activation, 291
BNF, 17
booléens, 13, 26
bootstrap, 371
bouclage, 8
boucle d'interaction, 142, 183, 193, 222, 332, 358
boucles, 39, 41, 49, 51, 61, 68, 101, 129
bouton, 193

cadre, 194
Camélia, 125
CamlTk, 193
canaux d'entrée-sortie, 182, 183
caractéristique, 117
caractères, 24, 46
cartes, 115
cas inutiles dans le filtrage, 120
chaînes de caractères, 24, 25, 46, 66, 88, 129, 133
champ, 117
chargement de fichiers, 179
circulaires (types), 354
codage de Huffman, 241, 242, 245
commandes, 38
commentaires, 10, 229
compilateur indépendant, 6, 180
compilation, 181, 186, 188, 190, 222, 289, 366
complexité, 31, 33, 35, 81, 98
composition de fonctions, 70, 96
compression, 237, 241
compteur, 47, 48, 54
concaténation de chaînes, 8, 68, 88, 100
concaténation de listes, 84, 92, 97, 102
conditionnelle, 12, 26, 43, 84
`configure`, 195
conjonction, 26, 209
connecteurs propositionnels, 208
cons, 75
constructeurs de listes, 75
constructeurs de types, 63
constructeurs de valeurs, 110, 116
contrainte de type, 11, 62
contraintes de type, 11
conventions syntaxiques, 15
cosinus, 150
couleur de tracé, 152
crayon, 149, 155
curryfication, 64, 72

décalage logique, 251, 252, 263
déclarations, 187
déclenchement d'exceptions, 127, 172
décompression, 242
définition, 9
définition par cas, 28
définitions, 6, 8, 20
déréférencement, 47
déterminisation, 313
diagrammes syntaxiques, 17, 39, 42, 84, 129
disjonction, 26, 209
`do`, 39
`done`, 39
`downto`, 39

effacement d'un fichier, 239
effet, 13, 37, 53, 54
`else`, 12

`end`, 14
enregistrements, 116, 150
ensembles, 316, 320
entrée standard, 183
entrées-sorties, 181, 238, 334
entrées-sorties structurées, 246, 265, 274
environnements d'évaluation, 170, 172, 329
environnements de typage, 284
erreurs, 126
Ershov, 295
et (booléen), 26
et bit à bit, 252, 263
`exception`, 128
exceptions, 77, 126, 127, 129, 172, 215, 334
`exn`, 127, 128
exponentielle (complexité), 32, 360
expressions, 168
expressions rationnelles, 305
extensionnalité, 52

factorielle, 20, 49, 268, 275
factorisation à gauche, 309
`false`, 13
fermeture d'un canal, 182
fermetures, 327, 329, 334, 369
Fibonacci, 33, 277, 303
fichiers, 182
file d'attente, 245, 247
filtrage, 28, 76, 78, 84, 85, 110, 117, 120, 126, 128, 158, 162, 165, 328
filtrage exhaustif, 122
filtrage partiel, 122
filtres « ou », 135
fin de fichier, 182
fin de phrase, 6
flèche (type fonctionnel), 9
`float_of_string`, 196
flocon, 147, 154
flottants (nombres), 34, 149
flux, 161, 165, 183
fonctionnelle (programmation), 37, 268
fonctionnelles, 59–61, 64, 66, 70, 72, 73, 81, 220, 279
fonctionnelles sur les listes, 81, 91, 94, 96
fonctions, 8–10, 67, 68, 342
fonctions anonymes, 11, 68, 343
fonctions calculées, 60
`for`, 39
fractales, 147
`function`, 11

généralisation, 345, 356
gardes dans les filtres, 121
GC, 367
glissière, 194
graphisme, 148

hachage, 229
Hanoi (les tours de), 28, 88
heaps, 248
Huffman, 240

identificateurs, 7
identité, 58, 62
`if`, 12
impérative (programmation), 38, 101, 268
implémentations de modules, 187
implication, 210
`implose`, 92, 99–101
impression à l'écran, 22, 39, 67, 140, 182
impression formatée, 195, 196, 293
`in`, 8
incrémentation, 48
inférence de types, 6, 61, 339
instructions (du pico-processeur), 256
`int_of_float`, 195
interactivité, 6, 181, 366
interface homme-machine, 193
interfaces, 221
interfaces de modules, 186–188
interprétation, 326, 366
interruption, 142
invite, 6
itérateurs, 81, 82, 92, 94, 96, 105

lecture au clavier, 67, 141, 182
lecture d'une référence, 47
lecture dans un tableau, 41
lecture sur fichier, 182
Lempel-Ziv, 253
`let`, 7, 8

`let rec`, 21, 22
lexèmes, 160
liaisons, 7, 20
linéaire (complexité), 32, 101
linéarité du filtrage, 121
listes, 75–78, 83, 88, 130
listes d'association, 131, 134, 170, 230
logique, 207
Logo, 147, 169
longueur d'un tableau, 41, 42
longueur d'une chaîne, 24, 46, 100
longueur d'une liste, 93, 104

`mainLoop`, 194
`make_matrix`, 199
`match`, 84
matrice, 199
menus, 197
modifications physiques, 38, 150, 163, 363
modules, 184
monomorphe, 57
mot-clé, 217
motifs « ou », 163
motifs intervalles, 163
`mutable`, 150

n-uplets, 63
négation, 208
nil, 75
noms, 7
noms extérieurs, 184
noms qualifiés, 184
`not`, 214

occurrence (test d'), 354
`of`, 110
`open`, 184
`openTk`, 194
ou (booléen), 26
ou bit à bit, 251, 263
ou exclusif, 263
ouverts (modules), 184
ouverture d'un canal, 182, 239

`pack`, 194
paires, 63, 69
palindromes, 25
paradoxes, 208
paramètres de types, 58
Pascal, 277
peintures, 113
phrase Caml, 6
pico-processeur, 255, 289
pile (du pico-processeur), 259, 291
piles, 318
point courant, 148
point fixe, 334
polymorphisme, 57, 66, 70, 71, 76, 81, 340, 342, 368
polynômes, 40, 42, 43, 45, 83, 85, 109, 111
portée dynamique, 20, 177, 328
portée statique, 20, 177, 327
positionnement dans un fichier, 246
`prefix`, 95
`print_char`, 24
`printf`, 293
priorité (file d'attente avec), 245, 247
priorités, 218, 281, 308
procédures, 14, 38, 168
produit (types), 63, 116
programmes indépendants, 180
prompt, 6
propositions, 208, 213

quadratique (complexité), 32, 81, 101, 102

récupération d'exceptions, 126, 128, 142, 172
récupération de mémoire, 367
récurrence, 32, 33, 36, 103, 122
récursifs (types), 114
récursion, 19, 22, 27, 28, 30, 32, 77, 154, 175, 177, 330, 334, 348
références, 47, 90, 101, 119, 230
règle η, 52, 54, 55, 95, 363
`random`, 201
`rec`, 22
records, 116
redéfinition, 48
`ref`, 47
registres (allocation de), 294, 298
registres (du pico-processeur), 256
regular expressions, 305

remplissage de rectangles, 152
retard de l'évaluation, 55, 157
retournement d'une liste, 101
rien (valeur `()`), 14
RISC, 255, 258

séquence, 14, 17, 53, 81
schémas de types, 340, 341
simulateur (du pico-processeur), 260
sinus, 150
sommation (d'une formule), 61
somme (types), 110, 116
sortie d'erreur, 183
sortie standard, 183
sous-chaîne, 26, 134, 240
sous-programmes, 259
spécialisation, 59, 62, 67, 345, 357
spirales, 175
`sprintf`, 195, 196
stack frame, 291
streams, 161
suspensions, 334
synonymes (dans les filtres), 80, 117
syntaxe abstraite, 159, 278, 289, 365
syntaxe concrète, 159, 365
synthèse de types, 339, 344

tableaux, 40, 42, 65, 68, 75, 270
terminaison, 8
`then`, 12
Thompson, 312
Tk, 193
`to`, 39
tracé de lignes, 148, 151
trace, 23
tri, 65, 78, 81, 104
trigonométrie, 150, 151
`true`, 13
`try`, 126, 128
typage, 5, 61, 283
`type`, 110

unification, 345, 353, 362
`unit`, 14

vérité (tables de), 209, 210
vérité (valeurs de), 13, 208
`value`, 187
variables, 7, 48
variables de types, 340, 348
variables libres, 216
variables rémanentes, 120
`vect`, 40, 63
vecteurs, 40
vidage des canaux, 183

`when`, 121
`while`, 39
`with`, 84, 126, 128